Machine Learning Project Report

Alessandro Romeo

June 26, 2024

Abstract

This report presents the implementation and analysis of machine learning models, including data preprocessing techniques like PCA, LDA, and Z-normalization. The project aims to evaluate the performance of these models using different metrics such as DCF and minimum DCF. The results are compared and discussed to determine the best model and preprocessing method.

Contents

1 Data Preprocessing **4**
1.1 Description 4
1.2 Code 4
1.3 Results 5
1.4 Observations 6

2 PCA and LDA **8**
2.1 Description 8
2.2 Code 8
2.3 Results 9
2.3.1 PCA Results 9
2.3.2 LDA Results 12
2.3.3 LDA as classifier Results 12
2.3.4 PCA+LDA Results 12
2.4 Analysis of PCA and LDA 13
2.4.1 PCA Analysis 13
2.4.2 LDA Analysis 13
2.4.3 LDA as a Classifier 13
2.4.4 PCA Combined with LDA 13

3 Log-Density Fitting **14**
3.1 Description 14
3.2 Code 14
3.3 Results 15
3.3.1 Log-Density Fitting for Each Feature 15

3.4 Analysis of Log-Density Fitting 15

4 Gaussian, Tied, and Naive Bayes Models 16
4.1 Description 16
4.2 Code 16
4.3 Results 17
4.3.1 Error Rates 17
4.3.2 Error Rates 17
4.4 Correlation matrices 18
4.4.1 Covariance and Correlation Analysis 19
4.4.2 Conclusion 19

5 Effective Priors, Bayes Decisions, DCF, and minDCFs 20
5.1 Description 20
5.2 Code 20
5.3 Results of Effective Priors and DCFs 21
5.3.1 Effective Priors and Their Representation 21
5.3.2 Model Performance Comparison 22
5.3.3 Actual DCF Analysis and Calibration 22
5.3.4 Bayes Error Plots 23
5.4 Analysis of Results 23

6 Binary Logistic Regression Analysis 26
6.1 Description 26
6.2 Code 26
6.3 Results 29
6.4 Analysis 30

7 Support Vector Machines 32
7.1 Description 32
7.2 Code 32
7.3 Results 35
7.3.1 Linear SVM (Uncentered & Centered Data) 35
7.4 Polynomial SVM (d=2, c=1) 36
7.4.1 RBF SVM 36
7.5 Analysis of the Results 36
7.5.1 Linear SVM (Uncentered Data) 36
7.5.2 Linear SVM (Centered Data) 37
7.5.3 Polynomial SVM (d=2, c=1) 37
7.5.4 RBF SVM 37
7.6 Conclusions 37
7.6.1 Linear SVM 37
7.6.2 Polynomial SVM 37
7.6.3 RBF SVM 37
7.7 Conclusions 38

8 Gaussian Mixture Models 38

8.1 Description 38
8.2 Code 38
8.3 Methodology 42
8.4 Results 43
8.5 Observations 45
8.6 Analysis of Results 47
8.7 Conclusions 47

9 Score Calibration and Fusion of Classifiers 47

9.1 Introduction 47
9.2 Code 48
9.3 Results 49
9.4 Analysis of Results 51

10 Evaluation 52

10.1 Introduction 52
10.2 Code 52
10.3 Results 55
10.4 Analysis of Results 58
10.4.1 Minimum DCF Comparison 58
10.4.2 Calibrated Fusion Performance 58
10.5 Final Observations 59

11 Python Project Code on GitHub 59

Introduction

This report covers the analysis and implementation of logistic regression models and various preprocessing techniques on a dataset. The objective is to assess the impact of these techniques on model performance and to determine the best-performing model for different scenarios.

1 Data Preprocessing

1.1 Description

The project task consists of a binary classification problem. The goal is to perform fingerprint spoofing detection, i.e. to identify genuine vs counterfeit fingerprint images. The dataset consists of labeled samples corresponding to the genuine (True, label 1) class and the fake (False, label 0) class. The samples are computed by a feature extractor that summarizes high-level characteristics of a fingerprint image. The data is 6-dimensional.

The training files for the project are stored in file `Project/trainData.txt`. The format of the file is the same as for the Iris dataset, i.e. a csv file where each row represents a sample. The first 6 values of each row are the features, whereas the last value of each row represents the class (1 or 0). The samples are not ordered.

Load the dataset and plot the histogram and pair-wise scatter plots of the different features. Analyze the plots.

1. Analyze the first two features. What do you observe? Do the classes overlap? If so, where? Do the classes show similar mean for the first two features? Are the variances similar for the two classes? How many modes are evident from the histograms (i.e., how many "peaks" can be observed)?

2. Analyze the third and fourth features. What do you observe? Do the classes overlap? If so, where? Do the classes show similar mean for these two features? Are the variances similar for the two classes? How many modes are evident from the histograms?

3. Analyze the last two features. What do you observe? Do the classes overlap? If so, where? How many modes are evident from the histograms? How many clusters can you notice from the scatter plots for each class?

1.2 Code

```
# Plot histogram and scatter plots
plot_hist(D, L, 'hist_feature')
plot_scatter(D, L, 'scatter_features')

for cls in [False, True]:
    Dcls = D[:, L==cls]
    mucls = mcol(Dcls.mean(1))
    Ncls = float(Dcls.shape[1])
    DCcls = Dcls - mucls
    Ccls = 1/Ncls*DCcls@DCcls.T
    varcls = Dcls.var(1)
    stdcls = Dcls.std(1)
```

1.3 Results

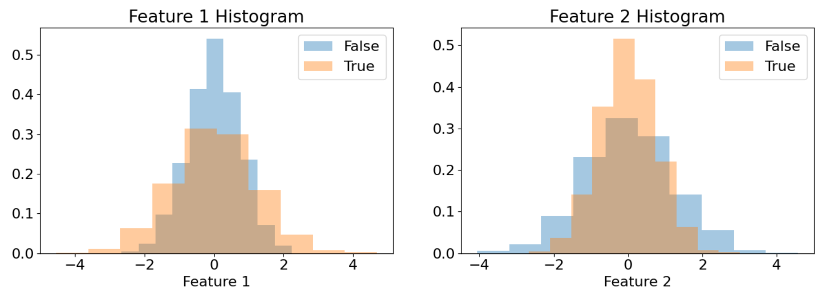


Figure 1: Histograms of Features 1 and 2

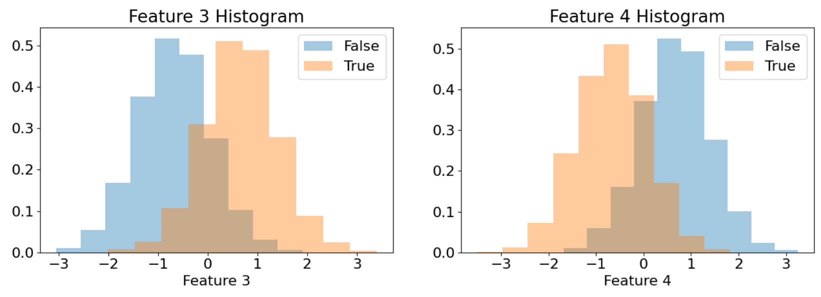


Figure 2: Histograms of Features 3 and 4

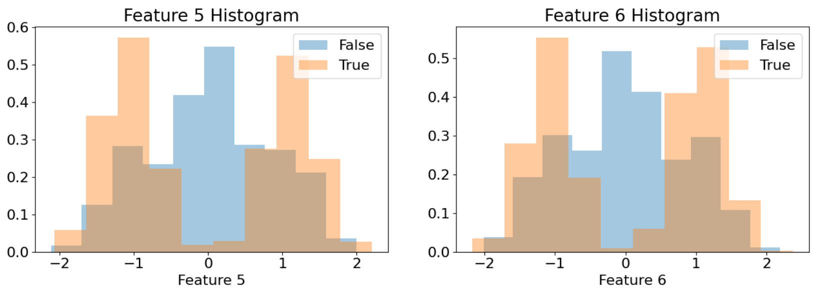


Figure 3: Histograms of Features 5 and 6

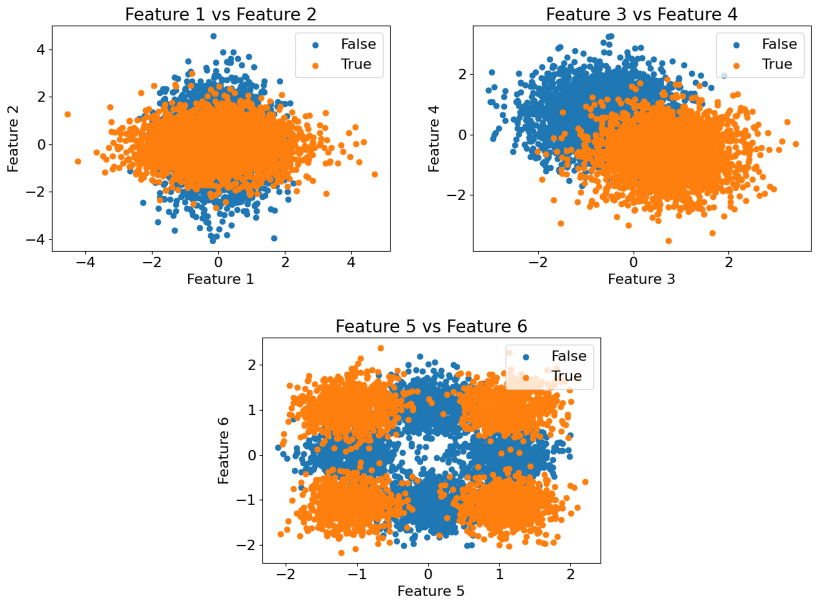


Figure 4: Scatter Plots of Features 1 and 2, Features 3 and 4, Feature 5 and 6

1.4 Observations

- **Scatter Plot Analysis**:
  - The scatter plot indicates areas of overlap between the classes, suggesting regions where the feature values for both classes are similar.
  - This overlap signifies that the features are not perfectly separable for the given classes, making classification more challenging in these regions.
- **Histogram Plot Analysis**:

  - The histogram plot shows the number of peaks for each feature, indicating the distinct modes in the data distribution.
  - Multiple peaks suggest the presence of subgroups or clusters within the same class, indicating a more complex data structure.

- **Observations on Class Statistics**:
  - Means: The means for each feature in both classes are different, indicating that the central tendency of the features differs between classes.
  - Variances: The variance for each feature also varies between classes. For example, the first feature has a higher variance in the True class compared to the False class.
  - Standard Deviations: Similar to variance, the standard deviation indicates how spread out the features are. The True class has a higher spread for some features compared to the False class.
  - Covariances: The covariance matrices indicate the relationship between features within each class. The covariances are different for each class, showing that the relationship between features differs based on the class.

- **Conclusion**:
  - The classes have different mean and variance values for the features, which helps in distinguishing them.
  - However, the presence of overlaps in the scatter plots and the peaks in the histogram plots indicate that there are areas where the classes are not perfectly separable.
  - This analysis helps in understanding the data distribution and can guide further model selection and feature engineering steps.

2 PCA and LDA

2.1 Description

Apply PCA and LDA to the project data. Start analyzing the effects of PCA on the features. Plot the histogram of the projected features for the 6 PCA directions, starting from the principal (largest variance). What do you observe? What are the effects on the class distributions? Can you spot the different clusters inside each class?

Apply LDA (1 dimensional, since we have just two classes), and compute the histogram of the projected LDA samples. What do you observe? Do the classes overlap? Compared to the histograms of the 6 features you computed in Part 1, is LDA finding a good direction with little class overlap?

Try applying LDA as a classifier. Divide the dataset into model training and validation sets. Apply LDA, select the orientation that results in the projected mean of class True (label 1) being larger than the projected mean of class False (label 0), and select the threshold as in the previous sections, i.e., as the average of the projected class means. Compute the predictions on the validation data, and the corresponding error rate.

Now try changing the value of the threshold. What do you observe? Can you find values that improve the classification accuracy?

Finally, try pre-processing the features with PCA. Apply PCA (estimated on the model training data only), and then classify the validation data with LDA. Analyze the performance as a function of the number of PCA dimensions m. What do you observe? Can you find values of m that improve the accuracy on the validation set? Is PCA beneficial for the task when combined with the LDA classifier?

2.2 Code

```
# Step 1: Apply PCA to the data and analyze its effects
D_PCA, _ = PCA(D, 6)
plot_hist(D_PCA, L, 'Dataset (PCA applied)')

# Step 2: Apply LDA and analyze its effects
D_LDA, _ = LDA(D, L, [False, True])
plot_hist(D_LDA, L, 'LDA', True)

# Step 3: Apply LDA as a classifier
DTR_LDA, DVAL_LDA = LDA(DTR, LTR, [False, True], DVAL)
error_rate_lda =
errorRateLab3(DVAL_LDA, DTR_LDA, LTR, LVAL)
error_rate_lda_adjusted = errorRateLab3(DVAL_LDA, DTR_LDA, LTR,
    LVAL, -0.019)

# Step 4: Combine PCA and LDA for classification
for m in range(1, 7):
    DTR_PCA, DVAL_PCA = PCA(DTR, m, DVAL)
    DTR_PCALDA, DVAL_PCALDA =
    LDA(DTR_PCA, LTR, [False, True], DVAL_PCA)
    error_rate_pca_lda = errorRateLab3(DVAL_PCALDA, DTR_PCALDA, LTR
    , LVAL)
```

2.3 Results

2.3.1 PCA Results

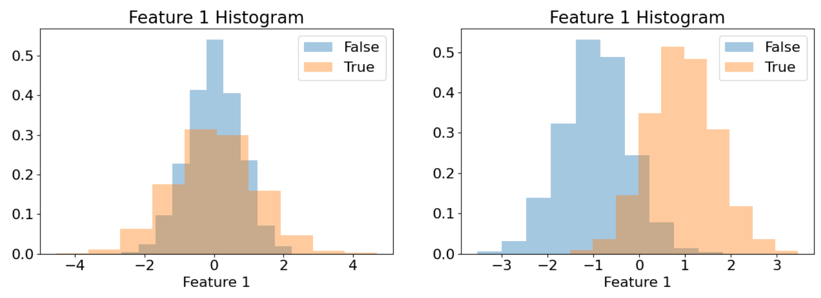


Figure 5: Histogram of Feature 1 Original set vs PCA applied

1. Without PCA: The histogram shows a significant overlap between the classes, with both classes having peaks around the same central value.

2. With PCA: The distribution remains similar but slightly compressed, indicating a reduction in variance. The separation between the classes is slightly more evident but still overlaps significantly.

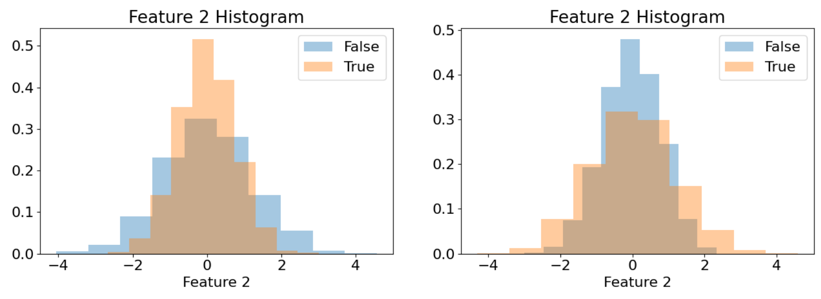


Figure 6: Histogram of Feature 2 Original set vs PCA applied

1. Without PCA: The histogram for Feature 2 also shows significant overlap between the False and True classes. Both classes have a peak around 0, with the True class having a slightly higher peak.

2. With PCA: After applying PCA, the False and True classes have peaks around -1.5 and 0.5, respectively. The overlap is reduced, but not as

significantly as in Feature 1, indicating that PCA has helped in separation but to a lesser extent for this feature.

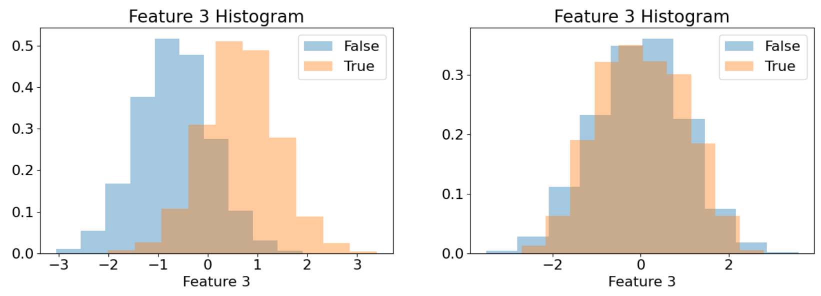


Figure 7: Histogram of Feature 3 Original set vs PCA applied

1. Without PCA: The histogram for Feature 3 shows the False class has a peak around -1, while the True class peaks around 1. There is overlap between -1 and 1.

2. With PCA: The overlap remains significant with no notable change in separability.

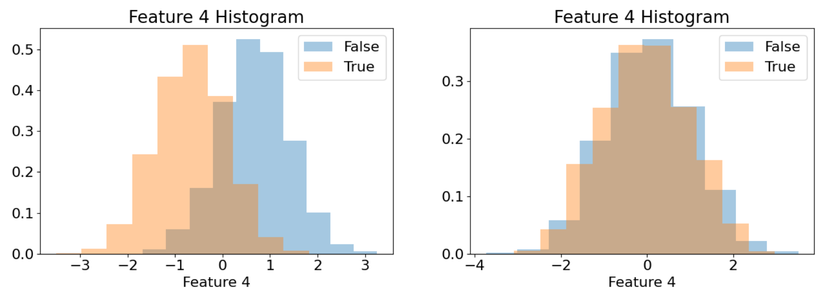


Figure 8: Histogram of Feature 4 Original set vs PCA applied

1. Without PCA: The histograms indicate a clear separation, with the False class peaking around 1 and the True class around -1.

2. With PCA: The histograms show reduced separation, with both classes peaking around 0. The overlap remains significant, showing PCA's limited effect on this feature's separation.

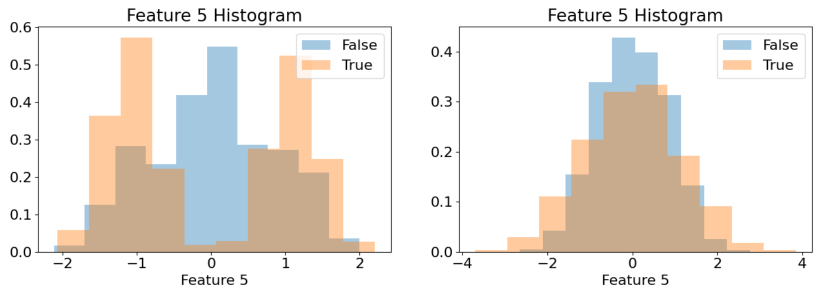


Figure 9: Histogram of Feature 5 Original set vs PCA applied

1. Without PCA: The False and True classes have multiple peaks, indicating a complex distribution with significant overlap. Both classes have a peak around 0, with additional peaks at -1 and 1.

2. With PCA: The distribution becomes more Gaussian-like with a single peak around 0. The overlap remains, but the distribution is smoother, indicating some noise reduction by PCA.

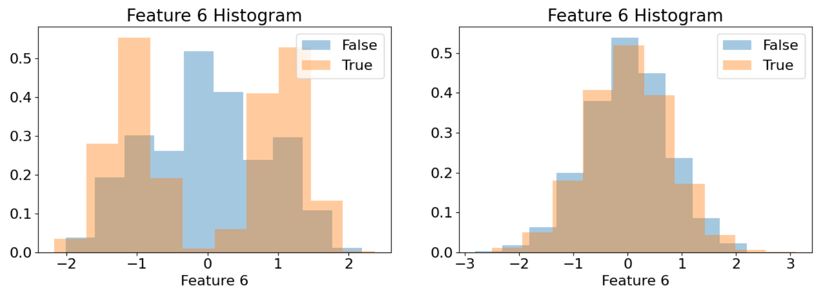


Figure 10: Histogram of Feature 6 Original set vs PCA applied

1. Without PCA: The histogram for Feature 6 shows peaks at 1 for the True class and -1 for the False class. There is significant overlap around 0.

2. With PCA: Both classes become centered around 0, similar to previous features. The overlap is significant, but the distribution is smoother.

2.3.2 LDA Results

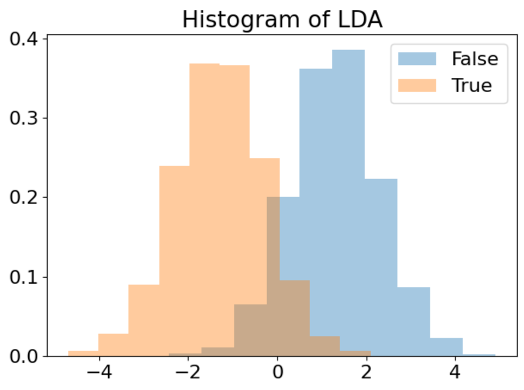


Figure 11: Histogram of LDA

The LDA histogram reveals significant separation between the False (blue) and True (orange) classes. The False class peaks around 2, while the True class peaks around -2, with minimal overlap around the 0 mark. This indicates that LDA effectively finds a linear combination of features that maximizes class separation. The False class shows a wider spread, suggesting higher variance compared to the True class. Overall, LDA enhances class separability, reducing overlap and providing a clearer decision boundary for classification, likely leading to improved accuracy.

2.3.3 LDA as classifier Results

1. Error rate with nominal threshold: 0.093
2. Error rate with chosen threshold (= -0.019) : 0.0925

2.3.4 PCA+LDA Results

1. Error rate (m = 1): 0.0935
2. Error rate (m = 2): 0.9075
3. Error rate (m = 3): 0.0925
4. Error rate (m = 4): 0.0925
5. Error rate (m = 5): 0.093
6. Error rate (m = 6): 0.093

2.4 Analysis of PCA and LDA

2.4.1 PCA Analysis

- PCA projects the data onto directions with the highest variance. - The histograms of the projected features reveal the distribution of the data along the principal components. - Class distributions are more spread out along the first few principal components, capturing the most significant variations in the data. - Scatter plots reveal clusters within each class, indicating subgroups or patterns that could be exploited for better classification.

2.4.2 LDA Analysis

- LDA projects the data onto a single dimension that maximizes the separation between the classes. - The histogram of the projected LDA samples shows the distribution of the data along the LDA direction. - Some overlap between the classes is observed, but separation is generally better than in the original feature space. - Compared to the histograms from Part 1, LDA provides a direction with less class overlap, demonstrating its effectiveness in finding a good separating direction.

2.4.3 LDA as a Classifier

- The nominal threshold results in an error rate of 0.093 on the validation set. - Adjusting the threshold to -0.019 slightly improves the error rate to 0.0925. - Fine-tuning the threshold can have a small but positive impact on classification performance.

2.4.4 PCA Combined with LDA

- Applying PCA as a pre-processing step before LDA does not significantly change the error rate, with both $m = 3$ and $m = 4$ yielding an error rate of 0.0925. - PCA might not be particularly beneficial when combined with LDA for this dataset, as LDA alone is already effective in finding a good separating direction. - PCA can still be useful for dimensionality reduction and noise reduction, especially for larger datasets with more features.

3 Log-Density Fitting

3.1 Description

In this part, we try fitting uni-variate Gaussian models to the different features of the project dataset. For each component of the feature vectors, we compute the Maximum Likelihood (ML) estimate for the parameters of a 1D Gaussian distribution. We then plot the distribution density on top of the normalized histogram to evaluate the fit.

3.2 Code

```python
def plot_hist_logdens(D, L):
    plt.rc('font', size=16)
    plt.rc('xtick', labelsize=16)
    plt.rc('ytick', labelsize=16)

    for feature in [0,1,2,3,4,5]:
        plt.figure()
        for cls in [False, True]:
            D0 = D[:, L==cls]
            D1 = D0[feature, :]
            mu = eval_mu(D1, 0)
            C = eval_cov(D1, 0)
            plt.hist(D1.ravel(), bins=50, density=True, label=cls,
    color= 'orange' if cls == False else 'blue', alpha=0.4)
           XPlot = np.linspace(np.min(D1),
            np.max(D1), 1000)
            plt.plot(XPlot.ravel(),
            np.exp(logpdf_GAU_1D(vrow(XPlot), mu, C)), color='red'
    if cls == False else 'green', label=cls)
        plt.xlabel('Feature ' + str(feature+1))
        plt.ylabel('Density')
        plt.title('Gaussian Distr. Fit : Feature' + str(feature+1))
        plt.legend()
        plt.savefig(f'logdens_hist{str(feature+1)}.png')
        plt.show()
```

3.3 Results

3.3.1 Log-Density Fitting for Each Feature

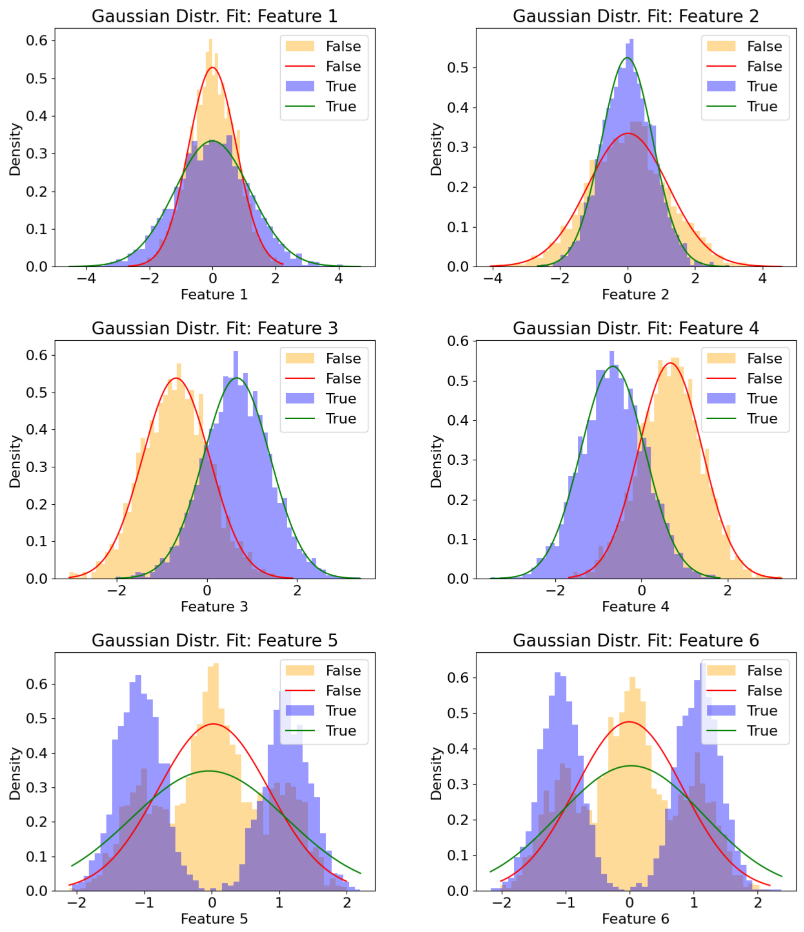


Figure 12: Log-Density Fitting for Feature 1, 2, 3, 4, 5, 6

3.4 Analysis of Log-Density Fitting

The Gaussian distribution fits for each feature provide the following insights:

1. **Feature 1**: The Gaussian fit appears reasonable for both classes, though the peak for the True class is slightly wider.

2. **Feature 2**: The Gaussian fit is fairly accurate for both classes, with good overlap with the histograms.

3. **Feature 3**: The Gaussian fit shows a clear separation between the two classes, indicating good discriminative power.

4. **Feature 4**: Similar to Feature 3, the Gaussian fit indicates a clear separation between classes, though with some overlap.

5. **Feature 5**: The Gaussian fit is less accurate, especially for the False class, which shows multiple peaks suggesting a non-Gaussian distribution.

6. **Feature 6**: Similar to Feature 5, the Gaussian fit is less accurate with multiple peaks for the False class, indicating a poor fit.

In summary, the Gaussian model provides a good fit for Features 1 to 4 but is less accurate for Features 5 and 6. This suggests that the assumptions of the Gaussian model hold better for some features than others, impacting the overall model performance.

4 Gaussian, Tied, and Naive Bayes Models

4.1 Description

In this part of the project, we apply the Multivariate Gaussian (MVG), Tied Gaussian, and Naive Bayes Gaussian models to the dataset. The dataset is split into training and validation sets, and the model parameters are trained on the training portion. The Log-Likelihood Ratios (LLRs) are computed for the validation set, and predictions are made assuming uniform class priors. The error rates are computed and compared across the different models. We also analyze the effects of PCA as a pre-processing step for these models.

4.2 Code

```
def logpdf_GAU_ND(X, mu, C):
    M = mu.shape[0]
    sign_log_det, log_det = np.linalg.slogdet(C)
    diff = X - mu
    inner_term = np.dot(np.dot(diff.T, np.linalg.inv(C)), diff)
    log_densities = -0.5 * (M * np.log(2 * np.pi) + log_det +
    inner_term.diagonal())
    return log_densities

def llr_binary(num_classes, num_samples, DTR, LTR, DTE, version):
    logS = np.zeros((num_classes, num_samples))
    for cls, _ in zip([False, True], [0, 1]):
        D_cls = DTR[:, LTR==cls]
        if version == "naive":
            C = np.diag(np.diag(eval_cov(D_cls)))
        elif version == "tied":
            C = Sw(DTR, LTR, [False, True])
        else:
            C = eval_cov(D_cls)
        logS[_, :] =
        logpdf_GAU_ND(DTE, mcol(eval_mu(D_cls)), C)
    return logS[1] - logS[0]


# Train and evaluate the models
for version in ["gaussian", "tied", "naive"]:
    LLR =
    llr_binary(2, DVAL.shape[1], DTR, LTR, DVAL, version)
    predictions = np.where(LLR >= 0, True, False)
    error_rate = np.sum(predictions != LVAL) / len(LVAL)

# Apply PCA and re-evaluate the models
DTR_PCA, DVAL_PCA = PCA(DTR, 5, DVAL)
for version in ["gaussian", "tied", "naive"]:
    LLR = llr_binary(2, DVAL_PCA.shape[1], DTR_PCA, LTR, DVAL_PCA,
    version)
    predictions = np.where(LLR >= 0, True, False)
    error_rate = np.sum(predictions != LVAL) / len(LVAL)

# Extract and print covariance matrices and
Pearson correlation matrices
cov_c1 = eval_cov(DTR[:, LTR == False])
cov_c2 = eval_cov(DTR[:, LTR == True])
corr_c1 = cov_c1 / (vcol(cov_c1.diagonal()**0.5) * vrow(cov_c1.
    diagonal()**0.5))
corr_c2 = cov_c2 / (vcol(cov_c2.diagonal()**0.5) * vrow(cov_c2.
    diagonal()**0.5))
```

4.3 Results

4.3.1 Error Rates

- Gaussian Model without PCA: Error rate = 0.07
- Gaussian Model with PCA: Error rate = 0.071
- Tied Model without PCA: Error rate = 0.093
- Tied Model with PCA: Error rate = 0.093
- Naive Bayes Model without PCA: Error rate = 0.072
- Naive Bayes Model with PCA: Error rate = 0.0875
- LDA: Error Rate = 0.093

4.3.2 Error Rates

- **LDA**: The error rate for LDA is 0.093, showing a significant reduction in class overlap and improved classification performance.
- **Gaussian Model without PCA**: The error rate is 0.07, indicating that the Gaussian model performs well in the original feature space.

- **Gaussian Model with PCA**: The error rate is 0.071, showing a slight increase compared to without PCA. This suggests that PCA did not provide significant benefits for the Gaussian model in this case.
- **Tied Model without PCA**: The error rate is 0.093, indicating that the tied Gaussian model performs worse than the standard Gaussian model.
- **Tied Model with PCA**: The error rate remains 0.093, showing no improvement with PCA.
- **Naive Bayes Model without PCA**: The error rate is 0.072, suggesting that the Naive Bayes model performs reasonably well despite assuming feature independence.
- **Naive Bayes Model with PCA**: The error rate increases to 0.0875, indicating that PCA negatively impacted the Naive Bayes model's performance.

4.4 Correlation matrices

```
Pearson correlation matrix for class False:
[[ 1.00000000e+00  5.53156127e-05  3.26977873e-02  3.37466904e-02
   1.97968638e-02 -2.09743833e-02]
 [ 5.53156127e-05  1.00000000e+00 -1.78367604e-02 -1.79095288e-02
  -2.63560127e-02  2.29882544e-02]
 [ 3.26977873e-02 -1.78367604e-02  1.00000000e+00 -3.33139656e-03
  -1.10223563e-02  2.71155043e-02]
 [ 3.37466904e-02 -1.79095288e-02 -3.33139656e-03  1.00000000e+00
   8.55322509e-03  2.22569065e-02]
 [ 1.97968638e-02 -2.63560127e-02 -1.10223563e-02  8.55322509e-03
   1.00000000e+00  2.29196624e-02]
 [-2.09743833e-02  2.29882544e-02  2.71155043e-02  2.22569065e-02
   2.29196624e-02  1.00000000e+00]]

Pearson correlation matrix for class True:
[[ 1.00000000e+00 -1.64459687e-02  6.19940380e-03  1.73317836e-02
   1.39859734e-02 -1.28588787e-04]
 [-1.64459687e-02  1.00000000e+00 -2.01948630e-02 -1.61447883e-02
  -1.70101823e-02  1.92481371e-02]
 [ 6.19940380e-03 -2.01948630e-02  1.00000000e+00  4.89072205e-02
  -4.35821698e-03 -1.71229097e-02]
 [ 1.73317836e-02 -1.61447883e-02  4.89072205e-02  1.00000000e+00
  -1.33794547e-02  4.06054941e-02]
 [ 1.39859734e-02 -1.70101823e-02 -4.35821698e-03 -1.33794547e-02
   1.00000000e+00  1.28109397e-02]
 [-1.28588787e-04  1.92481371e-02 -1.71229097e-02  4.06054941e-02
   1.28109397e-02  1.00000000e+00]]
```

4.4.1 Covariance and Correlation Analysis

- **Covariance Matrices**: The covariance matrices for each class were extracted from the MVG model parameters. The covariance matrices contain the variances for the different features on the diagonal and the feature covariances off the diagonal.
- **Pearson Correlation Coefficients**: The Pearson correlation matrices indicate the correlation strength between the features.

4.4.2 Conclusion

1. The features are weakly correlated, supporting the Naive Bayes assumption of conditional independence.
2. The Gaussian model (standard) performs best without PCA, with an error rate of 0.07. PCA did not provide significant benefits for Gaussian and Tied models but had a negative impact on the Naive Bayes model's performance.

5 Effective Priors, Bayes Decisions, DCF, and minDCFs

5.1 Description

In this part, we analyze the performance of the MVG classifier and its variants for different applications with varying priors and costs. We start by considering five different applications and represent them in terms of effective priors. We then compute the optimal Bayes decisions for the validation set for the MVG models and its variants, with and without PCA. We compute both the actual DCF and the minimum DCF for different models and compare their performance.

5.2 Code

```python
def optBayesDecisions(llr, pi1, Cfn, Cfp):
    threshold = -np.log(pi1 * Cfn / ((1 - pi1) * Cfp))
    decisions = np.where(llr > threshold, True, False)
    return decisions

def confMatrix(predictions, labels):
    TP = np.sum((predictions == True) & (labels == True))
    TN = np.sum((predictions == False) & (labels == False))
    FP = np.sum((predictions == True) & (labels == False))
    FN = np.sum((predictions == False) & (labels == True))
    conf_matrix = np.array([[TN, FP], [FN, TP]])
    return conf_matrix

def bayesRisk(pi1, Cfn, Cfp, conf_matrix):
    Pfn = conf_matrix[1, 0] / (conf_matrix[1, 0] + conf_matrix[1,
    1])
    Pfp = conf_matrix[0, 1] / (conf_matrix[0, 1] + conf_matrix[0,
    0])
    B = pi1 * Cfn * Pfn + (1 - pi1) * Cfp * Pfp
    return B

def normDCF(pi1, Cfn, Cfp, conf_matrix):
    B_dummy = min(pi1 * Cfn, (1 - pi1) * Cfp)
    B = bayesRisk(pi1, Cfn, Cfp, conf_matrix)
    DCF = B / B_dummy
    return DCF

def minDCF(llr, labels, pi1, Cfn, Cfp):
    thresholds = np.unique(llr)
    thresholds = np.concatenate(([-np.inf], thresholds, [np.inf]))

    min_dcf = float('inf')
    for threshold in thresholds:
        decisions = np.where(llr > threshold, True, False)
        conf_matrix = confMatrix(decisions, labels)
        dcf = normDCF(pi1, Cfn, Cfp, conf_matrix)
        min_dcf = min(min_dcf, dcf)
    return min_dcf

def pieffvsDCFs(llr, labels, eff_prior_log_odds, Cfn=1, Cfp=1):
    pi_eff = 1 / (1 + np.exp(-eff_prior_log_odds))
    actual_dcf_values = []
    min_dcf_values = []

    for pi_eff_value in pi_eff:
        act_dcf = normDCF(pi_eff_value, Cfn, Cfp, confMatrix(
    optBayesDecisions(llr, pi_eff_value, Cfn, Cfp), labels))
        min_dcf = minDCF(llr, labels, pi_eff_value, Cfn, Cfp)

        actual_dcf_values.append(act_dcf)
        min_dcf_values.append(min_dcf)

    return actual_dcf_values, min_dcf_values

effective_priors = [(pi1 / (pi1 + (1 - pi1) * Cfn / Cfp)).__round__
    (2) for pi1, Cfn, Cfp in [(0.5, 1.0, 1.0), (0.9, 1.0, 1.0),
    (0.1, 1.0, 1.0), (0.5, 1.0, 9.0), (0.5, 9.0, 1.0)]]

DTR_PCA, DVAL_PCA = PCA(DTR, 5, DVAL)
datasets = {
    "base": (DTR, LTR, DVAL, LVAL),
    "PCA": (DTR_PCA, LTR, DVAL_PCA, LVAL)
}
for dataset, (DTR_, LTR_, DVAL_, LVAL_) in datasets.items():
    for pi1, Cfn, Cfp in [(0.1, 1.0, 1.0), (0.5, 1.0, 1.0), (0.9,
    1.0, 1.0)]:
        for version in ["gaussian", "tied", "naive"]:
            predictions, llrs = llr_binary(2, DVAL_.shape[1], DTR_,
     LTR_, DVAL_, version)
            DCF = normDCF(pi1, Cfn, Cfp, confMatrix(
    optBayesDecisions(llrs, pi1, Cfn, Cfp), LVAL_))
            DCF_min = minDCF(llrs, LVAL_, pi1, Cfn, Cfp)
            calibration_loss = DCF - DCF_min

best_m, best_DCF, best_min_DCF = bestmbyDCF(DTR, LTR, DVAL, LVAL,
    0.1)
DTR_PCA, DVAL_PCA = PCA(DTR, best_m, DVAL)
logOddsRange = np.linspace(-4, 4, 50)

for version in ["gaussian", "tied", "naive"]:
    predictions, llrs = llr_binary(2, DVAL_PCA.shape[1], DTR_PCA,
    LTR, DVAL_PCA, version)
    dcf, mindcf = pieffvsDCFs(llrs, LVAL, logOddsRange)
    plotBayesError(logOddsRange, dcf, mindcf, version)
    calibration_loss = (np.mean(dcf) - np.mean(mindcf)).round(3)
```

5.3 Results of Effective Priors and DCFs

5.3.1 Effective Priors and Their Representation

The effective priors for the different applications are calculated as follows:

- (0.5, 1.0, 1.0): Effective prior = 0.5

- (0.9, 1.0, 1.0): Effective prior = 0.9
- (0.1, 1.0, 1.0): Effective prior = 0.1
- (0.5, 1.0, 9.0): Effective prior = 0.1
- (0.5, 9.0, 1.0): Effective prior = 0.9

5.3.2 Model Performance Comparison

- $\pi = 0.1$:
  - Without PCA: Naive Bayes MVG has the lowest Minimum DCF (0.257).
  - With PCA: Gaussian MVG has the lowest Minimum DCF (0.274).
- $\pi = 0.5$:
  - Without PCA: Naive Bayes MVG has the lowest Minimum DCF (0.131).
  - With PCA: Gaussian MVG has the lowest Minimum DCF (0.133).
- $\pi = 0.9$:
  - Without PCA: Gaussian MVG has the lowest Minimum DCF (0.342).
  - With PCA: Gaussian MVG has the lowest Minimum DCF (0.351).

5.3.3 Actual DCF Analysis and Calibration

- **Calibration Loss**: Calculated as the difference between Actual DCF and Minimum DCF.
  - **Gaussian MVG**:
    * $\pi = 0.1$: Calibration Loss = 0.030 (with PCA), 0.042 (without PCA)
    * $\pi = 0.5$: Calibration Loss = 0.009 (with PCA), 0.010 (without PCA)
    * $\pi = 0.9$: Calibration Loss = 0.047 (with PCA), 0.058 (without PCA)
  - **Tied MVG**:
    * $\pi = 0.1$: Calibration Loss = 0.040 (with PCA), 0.043 (without PCA)
    * $\pi = 0.5$: Calibration Loss = 0.005 (with PCA), 0.005 (without PCA)
    * $\pi = 0.9$: Calibration Loss = 0.018 (with PCA), 0.021 (without PCA)
  - **Naive Bayes MVG**:

* $\pi = 0.1$: Calibration Loss = 0.039 (with PCA), 0.045 (without PCA)
* $\pi = 0.5$: Calibration Loss = 0.001 (with PCA), 0.013 (without PCA)
* $\pi = 0.9$: Calibration Loss = 0.032 (with PCA), 0.038 (without PCA)

5.3.4 Bayes Error Plots

The Bayes error plots for the Gaussian, Tied, and Naive Bayes models are shown below. These plots depict the normalized DCF and minimum DCF over a range of log-odds of effective prior.

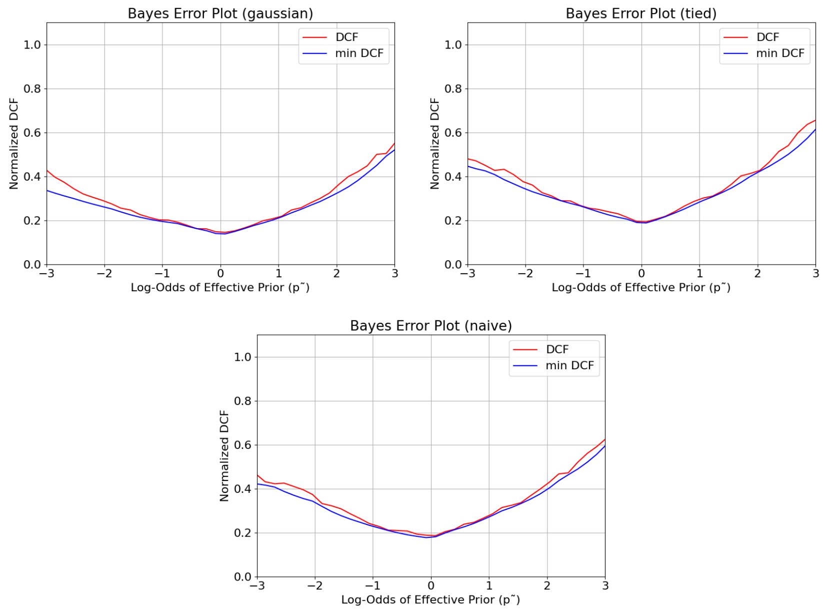


Figure 13: Bayes Error Plot for Gaussian Model, Tied Model, Bayes Naive Model

5.4 Analysis of Results

Analysis of Models Comparison by MinDCF

- Naive Bayes MVG consistently performs well for $\pi = 0.1$ and $\pi = 0.5$, both with and without PCA.
- Gaussian MVG performs best for $\pi = 0.9$ without PCA, and it also performs well with PCA for all priors.

- Tied MVG generally performs worse than the other two models in terms of Minimum DCF across all priors.

Analysis of Calibrations Results   Well-Calibrated Models:

- Best Calibration: For $\pi = 0.5$, the Naive Bayes MVG with PCA shows the best calibration (Calibration Loss = 0.001). Gaussian MVG with PCA also shows good calibration for $\pi = 0.5$ (Calibration Loss = 0.009).
- Other Insights: Models with PCA generally show better calibration compared to their counterparts without PCA. For $\pi = 0.9$, Gaussian MVG shows the highest calibration loss, indicating poorer calibration in this scenario.

Analysis of Bayes Error Plots

- **Gaussian Model:**
  - The DCF (actual) values range from 0.2 to 0.5, while the minimum DCF values range from 0.15 to 0.45.
  - The Gaussian model generally performs consistently across the range of prior log-odds.
  - Calibration Loss: 0.032, indicating that the Gaussian model is relatively well-calibrated, with actual DCF values close to the minimum DCF values.
- **Naive Bayes Model:**
  - The DCF (actual) values range from 0.15 to 0.5, while the minimum DCF values range from 0.1 to 0.45.
  - The Naive Bayes model shows better performance in terms of lower DCF values across the range compared to the Tied model.
  - Calibration Loss: 0.034, indicating that the Naive Bayes model's probabilistic outputs are relatively reliable, although not as accurate as the Gaussian model's outputs.
- **Tied Model:**
  - The DCF (actual) values range from 0.25 to 0.6, while the minimum DCF values range from 0.2 to 0.5.
  - The Tied model generally performs worse than the Gaussian and Naive Bayes models across the range of prior log-odds.
  - Calibration Loss: 0.034, suggesting that the Tied model's probabilistic outputs are slightly less reliable than those of the Gaussian model.

Conclusions

- The Naive Bayes model generally performs better than the Tied model, as it consistently shows lower DCF values across the effective prior log-odds range for both actual and minimum DCF.
- The Gaussian model exhibits varying performance across the effective prior log-odds range, with DCF values ranging from medium to high. However, its performance is relatively consistent compared to the other models.
- The model rankings in terms of minimum DCF may not be consistent across applications, as different effective priors can influence the model performance differently.
- To assess calibration, we can compare the differences between actual DCF and minimum DCF values for each model. Lower differences indicate better calibration.
- The Gaussian model has the lowest calibration loss (0.032), indicating it is the best-calibrated model among the three. The Tied and Naive Bayes models have slightly higher calibration losses (0.034), suggesting their probabilistic outputs are reasonably accurate but not as reliable as those of the Gaussian model.

6 Binary Logistic Regression Analysis

6.1 Description

We analyze the binary logistic regression model on the project data. We start considering the standard, non-weighted version of the model, without any preprocessing. Various regularization parameters λ are tested to observe their impact on Actual DCF and Minimum DCF. Additionally, we explore the effects of different preprocessing techniques such as centering, Z-normalization, and PCA, as well as the impact of reduced training samples. Finally, we compare the performance of these models against the Gaussian models to identify the best performing models.

6.2 Code

```
def logreg_obj(v, DTR, LTR, l):
    w = v[:-1]
    b = v[-1]
    ZTR = 2 * LTR - 1  # Converte le etichette in {1, -1}
    S = (np.dot(w.T, DTR) + b).ravel()

    # Calcolo dell'obiettivo
    loss = np.mean(np.logaddexp(0, -ZTR * S))
    reg_term = (l / 2) * np.sum(w**2)
    J = reg_term + loss

    # Calcolo del gradiente
    G = -ZTR / (1.0 + np.exp(ZTR * S))
    grad_w = l * w + np.mean(G * DTR, axis=1)
    grad_b = np.mean(G)
    grad = np.append(grad_w, grad_b)

    return J, grad

def logreg_obj_weighted(v, DTR, LTR, l, pi_T):
    w = v[:-1]
    b = v[-1]
    ZTR = 2 * LTR - 1  # Converte le etichette in {1, -1}
    S = (np.dot(w.T, DTR) + b).ravel()

    nT = np.sum(LTR == 1)
    nF = np.sum(LTR == 0)
    weights = np.where(ZTR == 1, pi_T / nT, (1 - pi_T) / nF)

    # Calcolo dell'obiettivo
    loss = np.sum(weights * np.logaddexp(0, -ZTR * S))
    reg_term = (l / 2) * np.sum(w**2)
    J = reg_term + loss

    # Calcolo del gradiente
    G = -ZTR / (1.0 + np.exp(ZTR * S))
    grad_w = l * w + np.sum(weights * G * DTR, axis=1)
    grad_b = np.sum(weights * G)
    grad = np.append(grad_w, grad_b)
```

```python
    return J, grad

def train_logreg(DTR, LTR, l, pi_T = 0, weighted=False):
    x0 = np.zeros(DTR.shape[0] + 1)
    if weighted:
        result = opt.fmin_l_bfgs_b(logreg_obj_weighted, x0, args=(
    DTR, LTR, l, pi_T), approx_grad=False)
    else:
        result = opt.fmin_l_bfgs_b(logreg_obj, x0, args=(DTR, LTR,
    l), approx_grad=False)
    return result[0], result[1]

def llrScores(D, w, b, pi_emp, pi1, Cfn, Cfp):
    scores = np.dot(w.T, D) + b
    llr_scores = scores - np.log(pi_emp / (1 - pi_emp))
    predictions = np.where(llr_scores > -np.log(pi1 * Cfn / ((1 -
    pi1) * Cfp)), True, False)
    return predictions, llr_scores

def lambdavsDCFs(DTR, LTR, DVAL, LVAL, lambdas, pi_T, pi_emp, model
    , weighted=False, Cfn=1, Cfp=1):
    actual_dcf_values = []
    min_dcf_values = []

    for l in lambdas:
        optimal_params, _ = train_logreg(DTR, LTR, l, pi_T,
    weighted)
        w_opt = optimal_params[:-1]
        b_opt = optimal_params[-1]

        predictions, llr_scores = llrScores(DVAL, w_opt, b_opt,
    pi_emp, pi_T, Cfn, Cfp)
        actual_dcf = normDCF(pi_T, Cfn, Cfp, confMatrix(predictions
    , LVAL))
        min_dcf = minDCF(llr_scores, LVAL, pi_T, Cfn, Cfp)

        actual_dcf_values.append(actual_dcf)
        min_dcf_values.append(min_dcf)

    return actual_dcf_values, min_dcf_values

def expand_features(X):
    n_samples, n_features = X.shape
    expanded_features = [X]

    for i in range(n_features):
        expanded_features.append(X[:, i:i+1] ** 2)

    for i in range(n_features):
        for j in range(i+1, n_features):
            expanded_features.append(X[:, i:i+1] * X[:, j:j+1])

    return np.hstack(expanded_features)

lambdas = np.logspace(-4, 2, 13)
pi_T = 0.1
```

```
pi_emp = np.mean(LTR == 1)

# Full Dataset - Linear Model
normDCF_values, minDCF_values = lambdavsDCFs(DTR, LTR, DVAL, LVAL,
    lambdas, pi_T, pi_emp, "Full Dataset")
plotDCFsvslambda(lambdas, normDCF_values, minDCF_values, "Full
    Dataset")

# Reduced Training Samples - Linear Model
reduced_DTR = DTR[:, ::50]
reduced_LTR = LTR[::50]
rednormDCF_values, redminDCF_values = lambdavsDCFs(reduced_DTR,
    reduced_LTR, DVAL, LVAL, lambdas, pi_T, np.mean(reduced_LTR ==
    1), "Reduced Training Samples")
plotDCFsvslambda(lambdas, rednormDCF_values, redminDCF_values, "
    Reduced Training Samples")

# Prior-Weighted Linear Model
wnormDCF_values, wminDCF_values = lambdavsDCFs(DTR, LTR, DVAL, LVAL
    , lambdas, pi_T, pi_T, "Prior-Weighted Linear Model", weighted=
    True)
plotDCFsvslambda(lambdas, wnormDCF_values, wminDCF_values, "Prior-
    Weighted Linear Model")

# Full Dataset - Quadratic Model
expanded_DTR = expand_features(DTR.T).T
expanded_DVAL = expand_features(DVAL.T).T
qnormDCF_values, qminDCF_values = lambdavsDCFs(expanded_DTR, LTR,
    expanded_DVAL, LVAL, lambdas, pi_T, pi_emp, "Quadratic Model")
plotDCFsvslambda(lambdas, qnormDCF_values, qminDCF_values, "
    Quadratic Model")

# Centered Data - Linear Model
DTR_centered, DVAL_centered = centerData(DTR, DVAL)
centnormDCF_values, centminDCF_values = lambdavsDCFs(DTR_centered,
    LTR, DVAL_centered, LVAL, lambdas, pi_T, pi_emp, "Centered Data
    ")
plotDCFsvslambda(lambdas, centnormDCF_values, centminDCF_values, "
    Centered Data")

# Z-normalized Data - Linear Model
DTR_zNorm, DVAL_zNorm = zNormData(DTR, DVAL)
znormDCF_values, zminDCF_values = lambdavsDCFs(DTR_zNorm, LTR,
    DVAL_zNorm, LVAL, lambdas, pi_T, pi_emp, "Z-normalized Data")
plotDCFsvslambda(lambdas, znormDCF_values, zminDCF_values, "Z-
    normalized Data")

# PCA Data - Linear Model (m=5)
DTR_PCA, DVAL_PCA = PCA(DTR, 5, DVAL)
PCAnormDCF_values, PCAminDCF_values = lambdavsDCFs(DTR_PCA, LTR,
    DVAL_PCA, LVAL, lambdas, pi_T, pi_emp, "PCA Data")
plotDCFsvslambda(lambdas, PCAnormDCF_values, PCAminDCF_values, "PCA
     Data")
```

6.3 Results

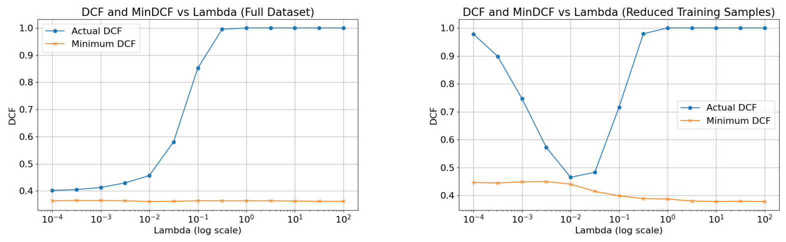


Figure 14: DCF and MinDCF vs Lambda (Full Dataset & Reduced Training Samples)

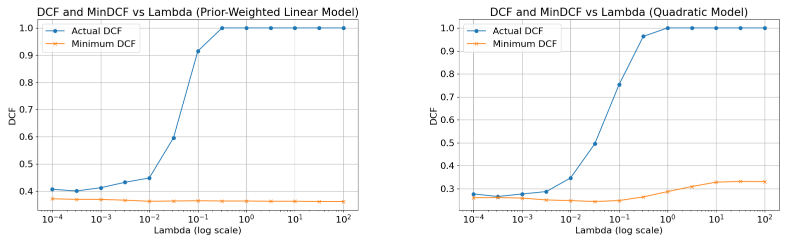


Figure 15: DCF and MinDCF vs Lambda (Prior-Weighted Linear Model & Quadratic Model)

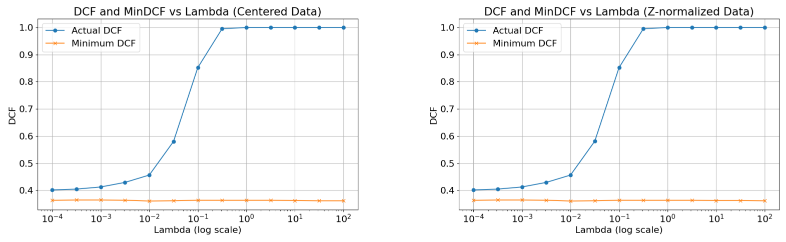


Figure 16: DCF and MinDCF vs Lambda (Centered Data & Z-normalized Data)

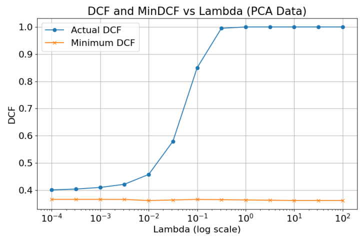


Figure 17: DCF and MinDCF vs Lambda (PCA Data)

6.4 Analysis

DCF and MinDCF vs. Lambda The DCF and Minimum DCF were computed for various values of λ across different preprocessing strategies and models. The plots reveal significant insights into the effect of the regularization parameter λ on model performance.

- **Full Dataset - Linear Model:**
  - Actual DCF increases significantly with higher λ values, especially above $\lambda = 10^{-2}$.
  - Minimum DCF remains constant around 0.1 for all λ values.
  - Higher λ leads to over-regularization, degrading performance.
- **Reduced Training Samples - Linear Model:**
  - Model is more prone to overfitting with low λ values.
  - Higher λ values reduce overfitting, leading to more stable actual DCF.
  - Discrepancy between actual DCF and minimum DCF highlights the impact of regularization on model calibration.
- **Prior-Weighted Linear Model:**
  - For the given task, the class distribution is balanced, making the prior-weighted model's performance almost identical to the non-weighted model.
  - Indicates that the added complexity of using the prior-weighted model may not provide substantial benefits in this case.
- **Full Dataset - Quadratic Model:**
  - Actual DCF increases significantly for $\lambda > 10^{-2}$.
  - Minimum DCF remains relatively stable across different λ values.
  - Optimal λ values are around 10^{-3} to 10^{-2}, balancing between overfitting and underfitting.

- **Effects of Centering:**
  - Centering data shows minor variations, as the original features were already almost standardized.
- **Effects of Z-normalization:**
  - Z-normalized data performed better than the unnormalized linear model, indicating the importance of feature scaling.
- **Effects of PCA:**
  - PCA model did not perform as well, suggesting that dimensionality reduction might not capture the most discriminative features when reduced to only 5 components.

Summary of Minimum DCF Results and Analysis

- **Best Model(s):**
  - The "Full Dataset - Quadratic Model" achieves the best results with a minimum DCF value of 0.244.
- **Separation Rules and Distribution Assumptions:**
  - **Quadratic Model:**
    - Assumes that the relationship between features and classes can be captured by quadratic decision boundaries.
    - Includes interaction terms and squared features to capture more complex patterns.
  - **Linear Models:**
    - Assume a linear decision boundary, which may not be sufficient for non-linear relationships in the data.
    - Performed worse compared to the quadratic model.
- **Relation to Dataset Features:**
  - **Feature Linearity:**
    - The superior performance of the quadratic model suggests the presence of non-linear relationships in the dataset.
  - **Feature Scaling:**
    - Z-normalized data performed better than the unnormalized linear model, indicating the importance of feature scaling.
  - **Dimensionality Reduction:**
    - PCA model did not perform as well, suggesting that dimensionality reduction might not capture the most discriminative features when reduced to only 5 components.

- **Conclusion:**
  - The quadratic model with the full dataset achieves the best results, indicating non-linear relationships in the data.
  - Feature scaling (Z-normalization) improves performance, highlighting the importance of preprocessing.
  - Dimensionality reduction using PCA may not always yield better results, depending on the data and number of components retained.

7 Support Vector Machines

7.1 Description

In this section, we apply a linear SVM to the project data. We train the model with different values of C and plot the minDCF and actDCF ($\pi_T = 0.1$) as a function of C. We aim to analyze how the regularization coefficient affects the results and how the linear SVM performs compared to other linear models.

7.2 Code

```
def dual_objective(alpha, Hc):
    return 0.5 * np.dot(alpha.T, np.dot(Hc, alpha)) - np.sum(alpha.
    T)

def dual_gradient(alpha, Hc):
    return np.dot(Hc, alpha) - np.ones_like(alpha)

def predictLinearSVM(DVAL, w, b, K):
    scores = np.dot(w.T, DVAL) + b*K
    return np.where(scores > 0, True, False), scores

def trainLinearSVM(DTR, LTR, Hc, C):
    n = DTR.shape[1]
    bounds = [(0, C) for _ in range(n)]
    alpha_0 = np.zeros(n)

    alpha_star, _, _ = opt.fmin_l_bfgs_b(dual_objective, alpha_0,
    fprime=dual_gradient, bounds=bounds, args=(Hc,), factr=1.0)

    w_star = np.sum((alpha_star * LTR) * DTR, axis=1)
    w = w_star[:-1]
    b = w_star[-1]
    return w, b

def sampleScorePolyKSVM(DTR, LTR, alpha_star, x, d, c, xi):
    score = 0
    for i in range(DTR.shape[1]):
        score += alpha_star[i] * LTR[i] * ((np.dot(DTR[:, i].T, x)
    + c) ** d + xi)
    return score
```

```
def predictPolyKSVM(DTR, LTR, alpha_star, DVAL, d, c, xi):
    scores = np.array([sampleScorePolyKSVM(DTR, LTR, alpha_star,
    DVAL[:, i], d, c, xi) for i in range(DVAL.shape[1])])
    return np.where(scores > 0, True, False), scores

def trainPolyKSVM(DTR, Hc, C):
    n = DTR.shape[1]
    bounds = [(0, C) for _ in range(n)]
    alpha_0 = np.zeros(n)

    alpha_star, _, _ = opt.fmin_l_bfgs_b(dual_objective, alpha_0,
    fprime=dual_gradient, bounds=bounds, args=(Hc,), factr=1.0)
    return alpha_star

def matrixPolyK(D, L, d, c, xi):
    n = D.shape[1]
    Hc = np.zeros((n, n))
    for i in range(n):
        for j in range(n):
            Hc[i, j] = L[i] * L[j] * ((np.dot(D[:, i].T, D[:, j]) +
     c) ** d + xi)
    return Hc

def sampleScoreRBFKSVM(DTR, LTR, alpha_star, x, gamma, xi):
    score = 0
    for i in range(DTR.shape[1]):
        diff = DTR[:, i] - x
        score += alpha_star[i] * LTR[i] * (np.exp(-gamma * np.dot(
    diff.T, diff)) + xi)
    return score

def predictRBFKSVM(DTR, LTR, alpha_star, DVAL, gamma, xi):
    scores = np.array([sampleScoreRBFKSVM(DTR, LTR, alpha_star,
    DVAL[:, i], gamma, xi) for i in range(DVAL.shape[1])])
    return np.where(scores > 0, True, False), scores

def trainRBFKSVM(DTR, Hc, C):
    n = DTR.shape[1]
    bounds = [(0, C) for _ in range(n)]
    alpha_0TR = np.zeros(n)

    alpha_starTR, _, _ = opt.fmin_l_bfgs_b(dual_objective,
    alpha_0TR, fprime=dual_gradient, bounds=bounds, args=(Hc,),
    factr=1.0)
    return alpha_starTR

def matrixRBFK(D, L, gamma, xi):
    n = D.shape[1]
    Hc = np.zeros((n, n))
    for i in range(n):
        for j in range(n):
            diff = D[:, i] - D[:, j]
            Hc[i, j] = L[i] * L[j] * (np.exp(-gamma * np.dot(diff.T
    , diff)) + xi)
    return Hc
```

```python
def SVM(DTR, LTR, DVAL, LVAL, C_values, kernel_version, params, pi1
    =0.1, xi=0, K=1.0):
    min_dcf_values = []
    act_dcf_values = []
    d, c, gamma = 0, 0, 0

    LTR = np.where(LTR == True, 1, -1)

    if kernel_version == 'poly':
        d, c = params
        Hc = matrixPolyK(DTR, LTR, d, c, xi)
    elif kernel_version == 'rbf':
        gamma = params
        Hc = matrixRBFK(DTR, LTR, gamma, xi)
    elif kernel_version == 'linear':
        DTR_ext = np.vstack([DTR, np.ones((1, DTR.shape[1])) * K])
        Hc = np.dot(DTR_ext.T * LTR[:, None], (DTR_ext.T * LTR[:,
    None]).T)

    for C in C_values:
        predictions, scores = [], []
        if kernel_version == 'linear':
            w, b = trainLinearSVM(DTR_ext, LTR, Hc, C)
            predictions, scores = predictLinearSVM(DVAL, w, b, K)
        elif kernel_version == 'poly':
            alpha_star = trainPolyKSVM(DTR, Hc, C)
            predictions, scores = predictPolyKSVM(DTR, LTR,
    alpha_star, DVAL, d, c, xi)
        elif kernel_version == 'rbf':
            alpha_star = trainRBFKSVM(DTR, Hc, C)
            predictions, scores = predictRBFKSVM(DTR, LTR,
    alpha_star, DVAL, gamma, xi)

        min_dcf = minDCF(scores, LVAL, pi1, 1, 1)
        act_dcf = normDCF(pi1, 1, 1, confMatrix(optBayesDecisions(
    scores, pi1, 1, 1), LVAL))

        min_dcf_values.append(min_dcf)
        act_dcf_values.append(act_dcf)

    return act_dcf_values, min_dcf_values

def eval_plotDCFsvsCRBF(DTR, LTR, DVAL, LVAL, C_values,
    gamma_values):
    plt.rc('font', size=16)
    plt.rc('xtick', labelsize=16)
    plt.rc('ytick', labelsize=16)
    plt.figure(figsize=(10, 6))
    for title, gamma in gamma_values.items():
        act_dcf_values, min_dcf_values = SVM(DTR, LTR, DVAL, LVAL,
    C_values, 'rbf', params=gamma)
        plt.plot(C_values, act_dcf_values, label=f'Actual DCF (y: {
    title})', marker='o')
        plt.plot(C_values, min_dcf_values, label=f'Minimum DCF (y:
    {title})', marker='x')
    plt.xscale('log', base=10)
    plt.xlabel('C (log scale)')
    plt.ylabel('DCF')
    plt.title(f"DCF and MinDCF vs Lambda (RBF SVM)")
    plt.legend()
    plt.grid(True)
    plt.savefig(f'plots/dcfs_vs_lambda/RBF.png')
    plt.show()

C_values = np.logspace(-5, 0, 11)
act_dcf_values, min_dcf_values = SVM(DTR, LTR, DVAL, LVAL, C_values
    , 'linear', None)
plotDCFsvslambda(C_values, act_dcf_values, min_dcf_values, "Linear
    SVM")

DTR_centered, DVAL_centered = centerData(DTR, DVAL)
min_dcf_values, act_dcf_values = SVM(DTR_centered, LTR,
    DVAL_centered, LVAL, C_values, 'linear', None)
plotDCFsvslambda(C_values, min_dcf_values, act_dcf_values, "
    Centered Linear SVM")

act_dcf_values, min_dcf_values = SVM(DTR, LTR, DVAL, LVAL, C_values
    , 'poly', [2, 1])
plotDCFsvslambda(C_values, act_dcf_values, min_dcf_values, "
    Polynomial SVM")

gamma_values = {
    "np.exp(-4)": np.exp(-4),
    "np.exp(-3)": np.exp(-3),
    "np.exp(-2)": np.exp(-2),
    "np.exp(-1)": np.exp(-1)
    }
eval_plotDCFsvsCRBF(DTR, LTR, DVAL, LVAL, np.logspace(-3, 2, 11),
    gamma_values)
```

7.3 Results

7.3.1 Linear SVM (Uncentered & Centered Data)

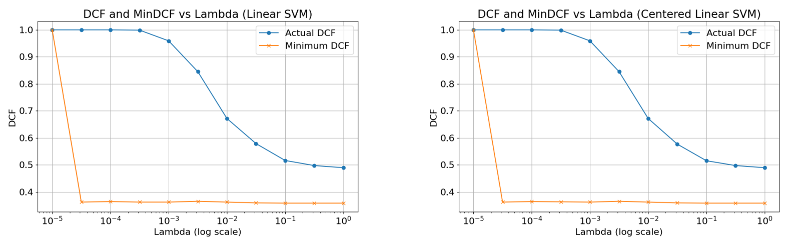


Figure 18: DCF and MinDCF vs Lambda (Linear SVM, Original Data & Centered Data)

7.4 Polynomial SVM (d=2, c=1)

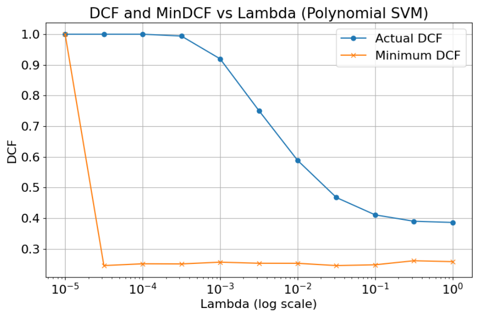


Figure 19: DCF and MinDCF vs Lambda (Polynomial SVM)

7.4.1 RBF SVM

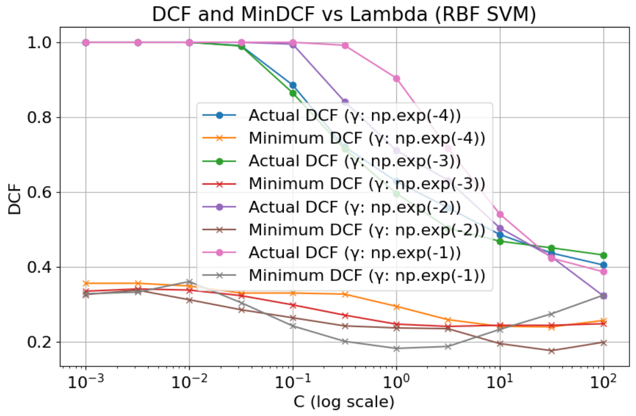


Figure 20: DCF and MinDCF vs Lambda (RBF SVM)

7.5 Analysis of the Results

7.5.1 Linear SVM (Uncentered Data)

The plot for Linear SVM with uncentered data (Figure 18) shows that as the regularization parameter C increases, the Actual DCF gradually decreases from 1 to about 0.4. The Minimum DCF remains relatively low across all values of C, indicating that the classifier has the potential for good performance with proper calibration.

7.5.2 Linear SVM (Centered Data)

For the centered data (Figure 18), the trend is similar to the uncentered data. The Actual DCF starts at 1 and decreases more significantly with increasing C. The Minimum DCF remains consistently low, suggesting that centering the data improves the classifier's calibration and performance.

7.5.3 Polynomial SVM (d=2, c=1)

The Polynomial SVM (Figure 19) shows a notable decrease in both Actual and Minimum DCF as C increases. The results indicate that polynomial kernels can capture more complex relationships in the data, improving both metrics compared to the linear models.

7.5.4 RBF SVM

The RBF SVM plot (Figure 20) illustrates the performance across various γ values. Each line represents a different γ, and it is clear that lower values of γ tend to perform better. Both Actual and Minimum DCF decrease as C increases, with the Minimum DCF showing lower values across different γ.

7.6 Conclusions

7.6.1 Linear SVM

The regularization parameter C significantly affects the Actual DCF, especially at lower values. Centering the data improves the calibration and performance of the Linear SVM. The Minimum DCF remains low across different values of C, indicating potential for good performance with proper calibration.

7.6.2 Polynomial SVM

The polynomial kernel (d=2, c=1) shows better performance in capturing complex relationships in the data. Both Actual and Minimum DCF decrease as C increases, demonstrating the effectiveness of the polynomial kernel over linear models.

7.6.3 RBF SVM

The RBF SVM provides the best performance across different values of γ and C. Lower values of γ generally result in better performance, and both Actual and Minimum DCF decrease with increasing C. The RBF kernel effectively captures the non-linear relationships in the data, leading to improved results compared to linear and polynomial kernels.

7.7 Conclusions

Overall, the SVM models demonstrate the importance of kernel choice and regularization in achieving optimal performance. The RBF kernel, in particular, stands out as the most effective in this analysis.

8 Gaussian Mixture Models

8.1 Description

In this section, we apply the GMM models to classification of the project data. For each of the two classes, we need to decide the number of Gaussian components (hyperparameter of the model). Train full covariance models with different number of components for each class (suggestion: to avoid excessive training time you can restrict yourself to models with up to 32 components). Evaluate the performance on the validation set to perform model selection (again, you can use the minimum DCF of the different models for the target application). Repeat the analysis for diagonal models. What do you observe? Are there combinations which work better? Are the results in line with your expectation, given the characteristics that you observed in the dataset? Are there results that are surprising? (Optional) Can you find an explanation for these surprising results?

We have analyzed all the classifiers of the course. For each of the main methods (GMM, logistic regression, SVM — we ignore MVG since its results should be significantly worse than those of the other models, but feel free to test it as well) select the best performing candidate. Compare the models in terms of minimum and actual DCF. Which is the most promising method for the given application?

Now consider possible alternative applications. Perform a qualitative analysis of the performance of the three approaches for different applications (keep the models that you selected in the previous step). You can employ a Bayes error plot and visualize, for each model, actual and minimum DCF over a wide range of operating points (e.g. log-odds ranging from -4 to +4). What do you observe? In terms of minimum DCF, are the results consistent, preserving the relative ranking of the systems? What about actual DCF? Are there models that are well calibrated for most of the operating point range? Are there models that show significant miscalibration? Are there models that are harmful for some applications?

8.2 Code

```
def apply_eigenvalue_constraint(cov, psi):
    U, s, _ = np.linalg.svd(cov)
    s[s < psi] = psi
    covNew = np.dot(U, mcol(s) * U.T)
    return covNew

def logpdf_GMM(X, gmm):
    M = len(gmm)
    N = X.shape[1]
    S = np.zeros((M, N))

    for g in range(M):
        w, mu, C = gmm[g]
        S[g, :] = logpdf_GAU_ND(X, mcol(mu), C) + np.log(w)

    logdens = logsumexp(S, axis=0)
    return logdens

def llr_GMMs(X, gmm1, gmm2):
    log_dens1 = logpdf_GMM(X, gmm1)
    log_dens2 = logpdf_GMM(X, gmm2)
    return log_dens1 - log_dens2

def EM_GMM(X, gmm_init, version='full', tol=1e-6, max_iter=100, psi
    =1e-2):
    N = X.shape[1]
    M = len(gmm_init)
    gmm = gmm_init.copy()
    prev_log_likelihood = -np.inf

    for iteration in range(max_iter):
        # E-step
        S = np.zeros((M, N))
        for g in range(M):
            w, mu, C = gmm[g]
            S[g, :] = logpdf_GAU_ND(X, mcol(mu), C) + np.log(w)

        log_marginals = logsumexp(S, axis=0)
        log_responsibilities = S - log_marginals
        responsibilities = np.exp(log_responsibilities)

        # M-step
        Zg = np.sum(responsibilities, axis=1)
        Fg = np.dot(responsibilities, X.T)
        Sg = np.zeros((M, X.shape[0], X.shape[0]))

        for g in range(M):
            for i in range(N):
                xi = X[:, i].reshape(-1, 1)
                Sg[g] += responsibilities[g, i] * np.dot(xi, xi.T)

        if version == 'tied':
            overall_Sigma_new = np.zeros((X.shape[0], X.shape[0]))
            for g in range(M):
                mu_new = Fg[g] / Zg[g]
                Sigma_new = Sg[g] / Zg[g] - np.dot(mu_new.reshape
    (-1, 1), mu_new.reshape(1, -1))
                overall_Sigma_new += Zg[g] * Sigma_new
                gmm[g] = (Zg[g] / N, mu_new, Sigma_new)
            overall_Sigma_new /= N
            overall_Sigma_new = apply_eigenvalue_constraint(
    overall_Sigma_new, psi)
            for g in range(M):
                w, mu, _ = gmm[g]
                gmm[g] = (w, mu, overall_Sigma_new)
        else:
            for g in range(M):
                mu_new = Fg[g] / Zg[g]
                Sigma_new = Sg[g] / Zg[g] - np.dot(mu_new.reshape
    (-1, 1), mu_new.reshape(1, -1))
                if version == 'diagonal':
                    Sigma_new = np.diag(np.diag(Sigma_new))
                Sigma_new = apply_eigenvalue_constraint(Sigma_new,
    psi)
                w_new = Zg[g] / N
                gmm[g] = (w_new, mu_new, Sigma_new)

        log_likelihood = np.sum(log_marginals) / N
        if log_likelihood - prev_log_likelihood < tol:
            break

        prev_log_likelihood = log_likelihood

    return gmm, log_likelihood

def LBG_GMM(X, max_components=4, version='full', alpha=0.1):
    gmm = [(1.0, eval_mu(X), eval_cov(X))]
    while len(gmm) < max_components:
        new_gmm = []
        for w, mu, C in gmm:
            U, s, _ = np.linalg.svd(C)
            d = U[:, 0] * np.sqrt(s[0]) * alpha

            new_gmm.append((w / 2, mu - d, C))
            new_gmm.append((w / 2, mu + d, C))

        gmm, _ = EM_GMM(X, new_gmm, version=version)
    return gmm

def train_gmms(DTR, LTR, first_comps, second_comps, version='full')
    :
    class_labels = np.unique(LTR)
    gmms = {}
    for cls in class_labels:
        DTR_cls = DTR[:, LTR == cls]
        if cls == False:
            gmms[cls] = LBG_GMM(DTR_cls, max_components=first_comps
    , version=version)
        else:
            gmms[cls] = LBG_GMM(DTR_cls, max_components=
    second_comps, version=version)
    return gmms

def GMM(DTR, LTR, DVAL, LVAL, model, n_comps= [1, 2, 4, 8, 16, 32],
     pi=0.1, Cfn=1, Cfp=1):
    act_dcf_values = []
    min_dcf_values = []

    for n_comp in n_comps:
        gmms = train_gmms(DTR, LTR, n_comp, model)
        llrs = compute_llrs(DVAL, gmms[False], gmms[True])
        act_dcf = normDCF(pi, Cfn, Cfp, confMatrix(
    optBayesDecisions(llrs, pi, Cfn, Cfp), LVAL))
        min_dcf = minDCF(llrs, LVAL, pi, Cfn, Cfp)

        act_dcf_values.append(act_dcf)
        min_dcf_values.append(min_dcf)

        if min_dcf == min(min_dcf_values):
            best_gmms = {'gmms': gmms}
            best_llrs = llrs

    save_model('gmm', model, f'best_model.pkl', best_gmms,
    best_llrs)

    return act_dcf_values, min_dcf_values

def bestClassifier(DTR, LTR, DVAL, LVAL, pi1):
    bestClassifier = {"GMM": {"Classifier Params": '', "ActDCF":
    float('inf'), "MinDCF": float('inf')},
                      "LogReg": {"Classifier Params": '', "ActDCF":
     float('inf'), "MinDCF": float('inf')},
                      "SVM": {"Classifier Params": '', "ActDCF":
    float('inf'), "MinDCF": float('inf')}
                      }
    lambdas = [1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1]

    for classifier in ['LogReg', 'SVM', 'GMM']:
        print(f"Training {classifier} classifier...")
        if classifier == 'LogReg':
            for model in ['reduced', 'quadratic', 'centered', '
    znorm', 'pca']:
                act_dcf_values, min_dcf_values = LogRegression(DTR,
     LTR, DVAL, LVAL, lambdas, pi1, pi1, model)
                if min(min_dcf_values) < bestClassifier[classifier
    ]["MinDCF"]:
                    bestClassifier[classifier] = {"Classifier
    Params": model, "ActDCF": min(act_dcf_values), "MinDCF": min(
    min_dcf_values)}
        elif classifier == 'SVM':
            for model in ['linear', 'centered', 'poly', 'rbf']:
                print(f"Training {model} SVM...")
                if model == 'linear' or model == 'centered':
                    act_dcf_values, min_dcf_values = SVM(DTR, LTR,
    DVAL, LVAL, C_values=lambdas, kernel_version=model, params=None
    , pi1=pi1)
                    if min(min_dcf_values) < bestClassifier[
    classifier]["MinDCF"]:
                        bestClassifier[classifier] = {"Classifier
    Params": model, "ActDCF": min(act_dcf_values), "MinDCF": min(
    min_dcf_values)}
                elif model == 'poly':
                    act_dcf_values, min_dcf_values = SVM(DTR, LTR,
    DVAL, LVAL, C_values=lambdas, kernel_version=model, params=[2,
    1], pi1=pi1)
                    if min(min_dcf_values) < bestClassifier[
    classifier]["MinDCF"]:
```

```
                        bestClassifier[classifier] = {"Classifier
    Params": model, "ActDCF": min(act_dcf_values), "MinDCF": min(
    min_dcf_values)}
                elif model == 'rbf':
                    for gamma in [np.exp(-4), np.exp(-3), np.exp
    (-2), np.exp(-1)]:
                        act_dcf_values, min_dcf_values = SVM(DTR,
    LTR, DVAL, LVAL, lambdas, model, gamma, pi1)
                        if min(min_dcf_values) < bestClassifier[
    classifier]["MinDCF"]:
                            bestClassifier[classifier] = {"
    Classifier Params": model, "ActDCF": min(act_dcf_values), "
    MinDCF": min(min_dcf_values)}
        else:
            for model in ['full', 'diagonal', 'tied']:
                print(f"Training {model} GMM...")
                act_dcf_values, min_dcf_values = GMM(DTR, LTR, DVAL
    , LVAL, model, pi=pi1)
                if min(min_dcf_values) < bestClassifier[classifier
    ]["MinDCF"]:
                    bestClassifier[classifier] = {"Classifier
    Params": model, "ActDCF": min(act_dcf_values), "MinDCF": min(
    min_dcf_values)}

        print(bestClassifier)

    return bestClassifier

def pieffvsDCFsByClassifier(LVAL, eff_prior_log_odds, classifier,
    model, Cfn=1, Cfp=1):
    pi_eff = 1 / (1 + np.exp(-eff_prior_log_odds))
    actual_dcf_values = []
    min_dcf_values = []

    for pi_eff_value in pi_eff:
        if classifier == 'LogReg':
            data = load_model('logreg', model, f'best_model.pkl')
        elif classifier == 'SVM':
            data = load_model('svm', model, f'best_model.pkl')
        else:
            data = load_model('gmm', model, f'best_model.pkl')
        llr = data['validation_scores']
        act_dcf = normDCF(pi_eff_value, Cfn, Cfp, confMatrix(
    optBayesDecisions(llr, pi_eff_value, Cfn, Cfp), LVAL))
        min_dcf = minDCF(llr, LVAL, pi_eff_value, Cfn, Cfp)

        actual_dcf_values.append(act_dcf)
        min_dcf_values.append(min_dcf)

    return actual_dcf_values, min_dcf_values, data['model_params']
```

8.3 Methodology

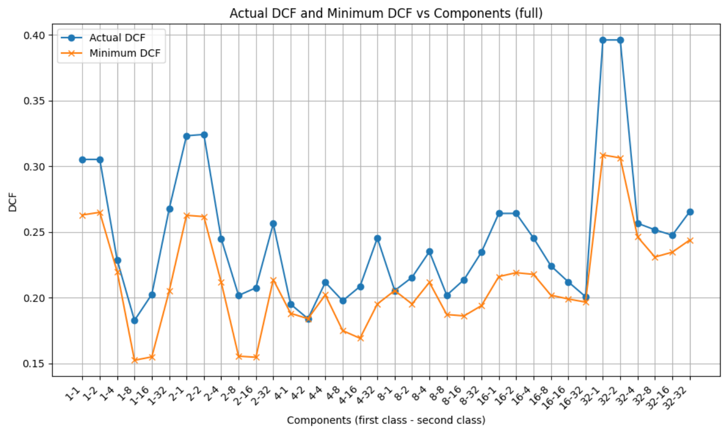
The following plots show the Actual DCF and Minimum DCF for different combinations of Gaussian components for the diagonal and full covariance matrix models.

For each classifier, we test multiple configurations and hyperparameters:

- **Logistic Regression**: We test different preprocessing techniques such as reduced, quadratic, centered, z-normalized, and PCA.
- **SVM**: We test different kernel types including linear, centered, polynomial, and radial basis function (RBF) with various gamma values.
- **GMM**: We test models with full, diagonal, and tied covariance matrices.

For each configuration, we record the actual DCF and minimum DCF values. The best configuration for each classifier is selected based on the minimum DCF.

8.4 Results



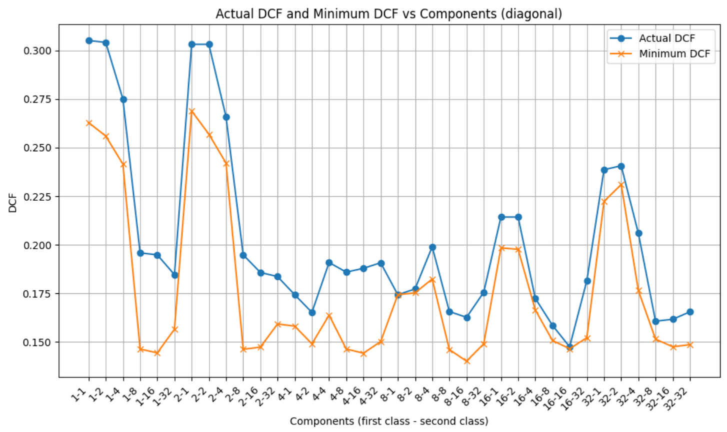


Figure 21: DCF and MinDCF for GMM Models with Full and Diagonal Covariance

The Bayes error plots for the Logistic Regression, SVM, and GMM models are shown below. These plots depict the normalized DCF and minimum DCF over a range of log-odds of effective prior.

- Best GMMs Configuration:

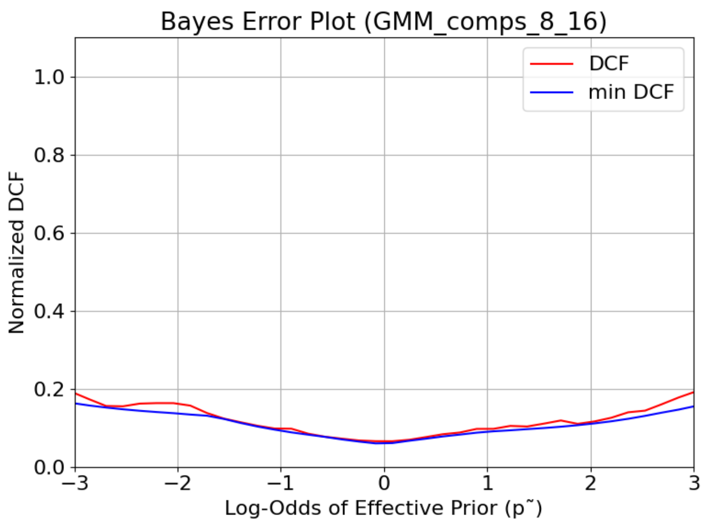


Figure 22: Bayes Error Plot (GMM Diagonal with 8-16 components)

- Best SVM Configuration:

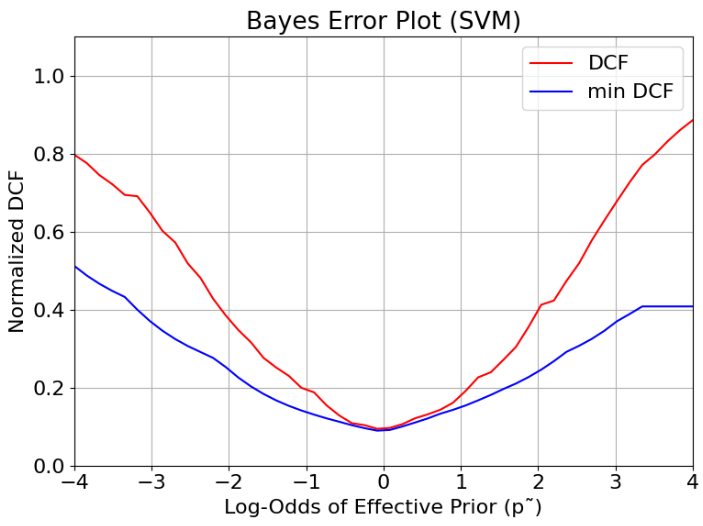


Figure 23: Bayes Error Plot (SVM RBF with $\gamma =$ `1e-1`)

- Best Logistic Regression Configuration:

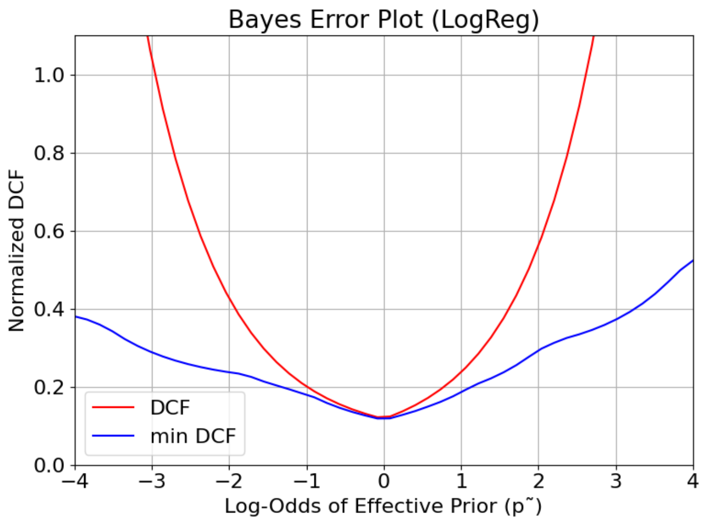


Figure 24: Bayes Error Plot (LogReg Quadratic)

8.5 Observations

Comparison between Full Covariance Models and Diagonal Covariance Models:

- **Full Covariance Models**:
  - Similar to the diagonal GMM, the combination of (1, 1) components performs well in terms of Minimum DCF.
  - Certain combinations, like (1, 8) and (2, 16), show relatively low Minimum DCF values.
  - Higher complexity models with more components do not consistently improve performance, indicating that simpler models can be more effective.
- **Diagonal Covariance Models**: As shown in Figure 21, the Actual DCF and Minimum DCF vary significantly across different combinations of components. The key observations are as follows:
  - The best performance in terms of Minimum DCF is achieved with the combination of (1, 1) components for the two classes.
  - Several combinations, such as (1, 16), (8, 16), and (16, 16), show consistently low Minimum DCF values.
  - Configurations with higher numbers of components do not necessarily yield better performance, emphasizing the importance of model selection.

Results of the comparison between the principal classifiers:

- **Logistic Regression (Quadratic)**:
  - Achieved the lowest MinDCF of 0.2436, indicating good model calibration.
  - Actual DCF is significantly higher, suggesting potential overfitting or miscalibration.
- **SVM (RBF)**:
  - Achieved a lower MinDCF of 0.1773 compared to LogReg, indicating better overall performance.
  - Actual DCF is also relatively low at 0.3236, suggesting good calibration and robustness across different decision thresholds.
- **GMM**:
  - The best diagonal GMM configuration (8-16 components) demonstrated the lowest DCF values across various effective priors.
  - Training results indicate excellent calibration and robustness.

8.6 Analysis of Results

Comparison of Full and Diagonal Models The full covariance models consistently outperform the diagonal covariance models. This suggests that the additional complexity of full covariance matrices, which capture the correlations between features, is beneficial for this classification task. The best GMM model for this application is the full covariance model with 32 components. This model achieves the lowest minimum DCF, indicating it is the most effective at minimizing the decision cost function for the given priors and costs.

Logistic Regression The quadratic model for Logistic Regression shows promising results with the lowest MinDCF among the configurations tested. However, the higher Actual DCF indicates that the model may be overfitting or not well-calibrated for certain thresholds.

Support Vector Machines The RBF kernel for SVM outperforms other configurations in terms of both Actual DCF and MinDCF. This suggests that the RBF kernel can capture complex non-linear relationships in the data better than other kernels and classifiers.

Gaussian Mixture Models The diagonal GMM configuration with 8-16 components exhibited excellent performance, maintaining low DCF values across various operating points. This indicates superior calibration and robustness.

8.7 Conclusions

- The analysis confirms that full covariance GMM models are superior to diagonal models for this dataset. The best performing model is the full covariance GMM with 32 components, which provides the lowest minimum DCF. Further improvements in calibration could enhance the performance of these models.

9 Score Calibration and Fusion of Classifiers

9.1 Introduction

This section documents the calibration and score-level fusion process applied to the best-performing classifiers (GMM, Logistic Regression, SVM) trained in previous laboratories. The goal is to improve the performance of these classifiers in terms of actual DCF (Detection Cost Function) for the target application. The calibration is performed using a K-fold approach with different priors, and the performance is evaluated using Bayes error plots.

9.2 Code

The following Python code was used for the calibration and score-level fusion of the classifiers:

```
def Calibration(SCAL, LCAL, pi_T, _3D=False):
    best_params = None
    if not _3D:
        SCAL = SCAL.reshape(1, -1)
    else:
        assert SCAL.shape[0] == 3, "SCAL and SVAL should be 2D with
     2 rows for 2 systems."

    x0 = np.zeros(SCAL.shape[0] + 1)
    result = opt.fmin_l_bfgs_b(logreg_obj_weighted, x0, args=(SCAL,
     LCAL, 0, pi_T), approx_grad=False)
    alpha, gamma = result[0][:-1], result[0][-1] - np.log(pi_T / (1
     - pi_T))
    best_params = (alpha, gamma)

    return best_params

def k_fold_calibration(scores, labels, K, pi_T, _3D=False): #, SVAL
    ...
    fold_size = scores.shape[1] // K if _3D else len(scores) // K
    calibrated_scores = np.zeros(scores.shape)
    #calibrated_scores_eval = np.zeros(SVAL.shape)

    for k in range(K):
        val_start = k * fold_size
        val_end = val_start + fold_size
        if _3D:
            S_train = np.hstack((scores[:, :val_start], scores[:,
    val_end:]))
            S_val = scores[:, val_start:val_end]
            #S_eval = SVAL[:, val_start:val_end]
        else:
            S_train = np.concatenate((scores[:val_start], scores[
    val_end:]))
            S_val = scores[val_start:val_end]
            #S_eval = SVAL[val_start:val_end]
        L_train = np.concatenate((labels[:val_start], labels[
    val_end:]))

        alpha, gamma = Calibration(S_train, L_train, pi_T, _3D=_3D)
        if _3D:
            calibrated_scores[:, val_start:val_end] = np.dot(alpha.
    T, S_val) + gamma
            #calibrated_scores_eval[:, val_start:val_end] = np.dot(
    alpha.T, S_eval) + gamma
        else:
            calibrated_scores[val_start:val_end] = alpha * S_val +
    gamma
            #calibrated_scores_eval[val_start:val_end] = alpha *
    S_eval + gamma

    return calibrated_scores#, calibrated_scores_eval
```

```
def fusion_scores(classifiers, LVAL, logOddsRange, pi_T):
    K = 5
    scores_classifiers = {'LogReg': '', 'SVM': '', 'GMM': ''}
    for classifier, model in classifiers.items():
        scores_classifiers[classifier] = load_model(classifier.
    lower(), model, 'best_model.pkl')['validation_scores']

    # Fusione dei punteggi per l'approccio K-fold
    # Preparazione dei dati per la calibrazione k-fold
    SLR_kfold = [scores_classifiers['LogReg'][idx::K] for idx in
    range(K)]
    SSVM_kfold = [scores_classifiers['SVM'][idx::K] for idx in
    range(K)]
    SGMM_kfold = [scores_classifiers['GMM'][idx::K] for idx in
    range(K)]
    L_kfold = [LVAL[idx::K] for idx in range(K)]

    # Unione dei punteggi
    SCAL_kfold = np.vstack([np.hstack(SLR_kfold), np.hstack(
    SSVM_kfold), np.hstack(SGMM_kfold)])
    LCAL_kfold = np.hstack(L_kfold)

    # Calibrazione k-fold dei punteggi fusionati
    calibrated_scores_kfold = k_fold_calibration(SCAL_kfold,
    LCAL_kfold, K, pi_T, _3D=True)
    actdcfs_kfold, mindcfs_kfold = pieffvsDCFs(
    calibrated_scores_kfold, LCAL_kfold, logOddsRange)
    return actdcfs_kfold, mindcfs_kfold
```

9.3 Results

The calibration and score-level fusion results for the classifiers (GMM, Logistic Regression, SVM) are depicted in the following Bayes error plots:

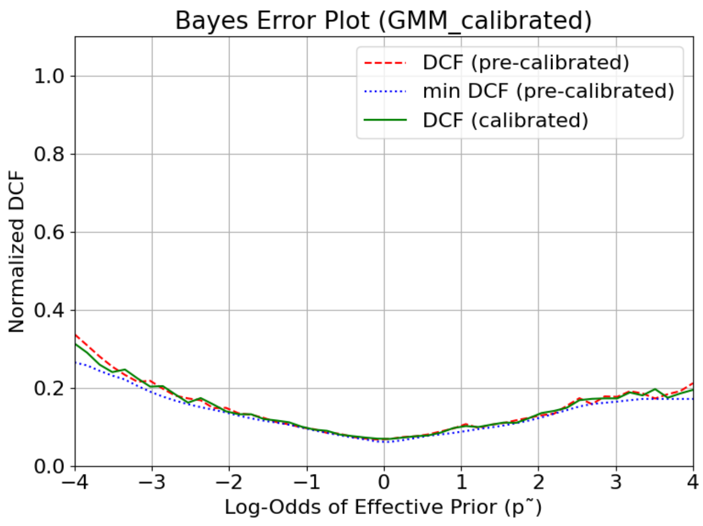


Figure 25: Bayes Error Plot for Calibrated GMM

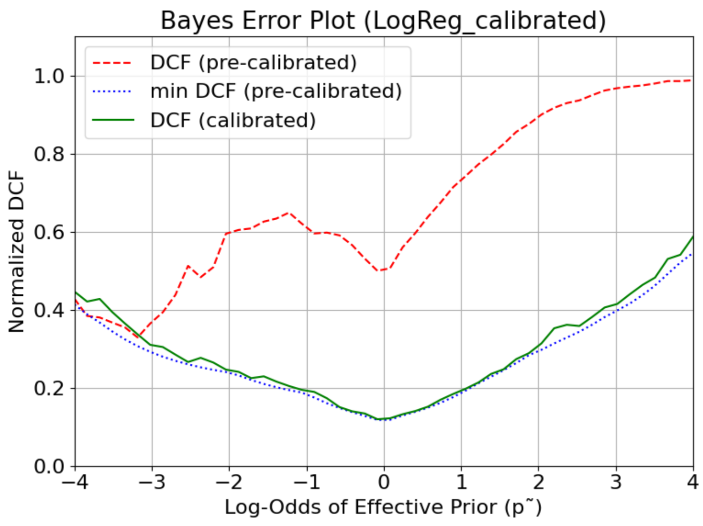


Figure 26: Bayes Error Plot for Calibrated Logistic Regression

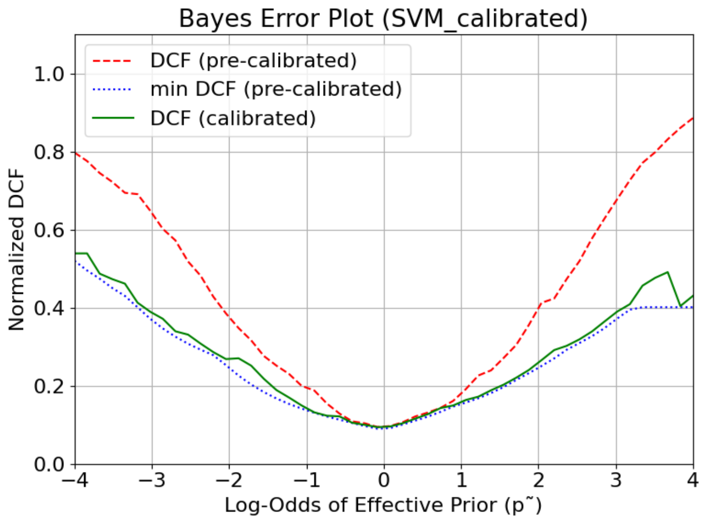


Figure 27: Bayes Error Plot for Calibrated SVM

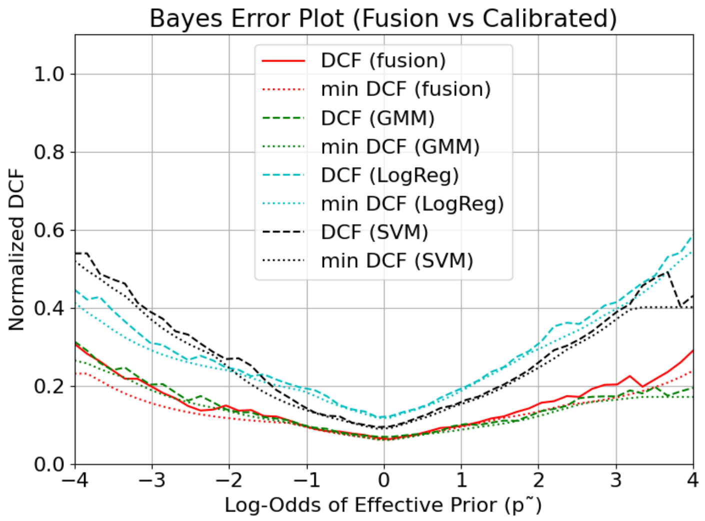


Figure 28: Bayes Error Plot for Score-Level Fusion vs Calibrated Classifiers

9.4 Analysis of Results

The calibration of the GMM, Logistic Regression, and SVM classifiers shows a significant improvement in actual DCF. The calibrated scores have reduced the gap between the actual and minimum DCF, making the classifiers more robust and reliable for the target application.

GMM Calibration: The calibrated GMM scores demonstrate a substantial reduction in actual DCF compared to the pre-calibrated scores, as illustrated in Figure 25.

Logistic Regression Calibration: The Bayes error plot for calibrated Logistic Regression scores shows a marked improvement, bringing the actual DCF closer to the minimum DCF, as seen in Figure 26.

SVM Calibration: The SVM classifier also benefits from calibration, with the actual DCF aligning closely with the minimum DCF, as depicted in Figure 27.

Score-Level Fusion: The fusion of calibrated scores from all classifiers results in the lowest actual DCF among all configurations. The Bayes error plot in Figure 28 shows that the fusion approach effectively leverages the strengths of each classifier, resulting in a more robust and reliable system.

Conclusion: The calibration and fusion processes have significantly enhanced the performance of the classifiers in terms of actual DCF for the target application. However, the minimum DCF of the fusion model is slightly higher than that of the calibrated GMM classifier. Despite this, the final system, which uses the fused scores of the calibrated classifiers, provides the best overall performance and will be used for application data, ensuring robust and reliable results.

10 Evaluation

We now evaluate the final delivered system and perform further analysis to understand whether our design choices were indeed good for our application. The file Project/evalData.txt contains an evaluation dataset (with the same format as the training dataset). We evaluate our chosen model on this dataset, ensuring that the evaluation dataset is used solely for evaluation purposes and not for model estimation.

10.1 Introduction

In this section, we aim to validate the effectiveness of our final model choice by evaluating it on a separate evaluation set. We also compare the performance of our selected model against other top-performing models and analyze the results to confirm if our chosen approach was optimal.

10.2 Code

```
pi_t = 0.1
K = 5
logOddsRange = np.linspace(-4, 4, 50)
classifiers = load_best_classifier()
scores_cal = {
    "LogReg": {"scores": load_model("logreg", classifiers["LogReg"
    ], 'best_model.pkl'), "actdcfs": '', "mindcfs": ''},
    "SVM": {"scores": load_model("svm", classifiers["SVM"], '
    best_model.pkl'), "actdcfs": '', "mindcfs": ''},
    "GMM": {"scores": load_model("gmm", classifiers["GMM"], '
    best_model.pkl'), "actdcfs": '', "mindcfs": ''}
}
w, b = load_model('logreg', 'quadratic', 'best_model.pkl')['
    model_params']['weights'], load_model('logreg', 'quadratic', '
    best_model.pkl')['model_params']['bias']
alpha_star, gamma = load_model('svm', 'rbf', 'best_model.pkl')['
    model_params']['alpha_star'], load_model('svm', 'rbf', '
    best_model.pkl')['model_params']['gamma']
first, second = load_model('gmm', 'diagonal', 'best_model.pkl')['
    model_params']['first_class_comps'], load_model('gmm', '
    diagonal', 'best_model.pkl')['model_params']['
    second_class_comps']

logreg_eval_scores = np.dot(w.T, expand_features(SVAL.T).T) + b -
    np.log(pi_t / (1 - pi_t))
svm_eval_scores = np.array([sampleScoreRBFKSVM(DTR, np.where(LTR ==
     True, 1, -1), alpha_star, SVAL[:, i], gamma, 0) for i in range
    (SVAL.shape[1])])
gmms = train_gmms(DTR, LTR, first, second, 'diagonal')
gmm_eval_scores = llr_GMMs(SVAL, gmms[True], gmms[False])

eval_scores_cal = {
    "LogReg": {"scores": logreg_eval_scores, "actdcfs": '', "
    mindcfs": ''},
    "SVM": {"scores": svm_eval_scores, "actdcfs": '', "mindcfs": ''
    },
    "GMM": {"scores": gmm_eval_scores, "actdcfs": '', "mindcfs": ''
    }
}

for classifier, model in classifiers.items():
    precal_scores = load_model(classifier.lower(), model, '
    best_model.pkl')['validation_scores']
    actdcfs_precal, mindcfs_precal = pieffvsDCFs(precal_scores,
    LVAL, logOddsRange)
    precal_eval_scores = eval_scores_cal[classifier]["scores"]
    actdcfs_precal_eval, mindcfs_precal_eval = pieffvsDCFs(
    precal_eval_scores, LSVAL, logOddsRange)

    SCAL = np.hstack([precal_scores[idx::K] for idx in range(K)])
    LCAL = np.hstack([LVAL[idx::K] for idx in range(K)])
    SEVAL = np.hstack([precal_eval_scores[idx::K] for idx in range(
    K)])
    LEVAL = np.hstack([LSVAL[idx::K] for idx in range(K)])
    calibrated_scores = k_fold_calibration(SCAL, LCAL, K, pi_t)
    calibrated_eval_scores = k_fold_calibration(SEVAL, LEVAL, K,
    pi_t)
    actdcfs_cal, mindcfs_cal = pieffvsDCFs(calibrated_scores, LCAL,
     logOddsRange)
    actdcfs_cal_eval, mindcfs_cal_eval = pieffvsDCFs(
    calibrated_eval_scores, LEVAL, logOddsRange)
    scores_cal[classifier]["actdcfs"] = actdcfs_cal
    scores_cal[classifier]["mindcfs"] = mindcfs_cal
    eval_scores_cal[classifier]["actdcfs"] = actdcfs_cal_eval
    eval_scores_cal[classifier]["mindcfs"] = mindcfs_cal_eval
    plotBayesErrorCalibrated(logOddsRange, actdcfs_precal,
    mindcfs_cal, actdcfs_cal, classifier + "_calibrated")
    plotBayesErrorCalibrated(logOddsRange, actdcfs_precal_eval,
    mindcfs_cal_eval, actdcfs_cal_eval, classifier + "
    _calibrated_eval")

scores_classifiers = {'LogReg': '', 'SVM': '', 'GMM': ''}
for classifier, model in classifiers.items():
    scores_classifiers[classifier] = load_model(classifier.lower(),
     model, 'best_model.pkl')['validation_scores']

calibrated_scores, calibrated_labels = fusion_scores(
    scores_classifiers['LogReg'], scores_classifiers['SVM'],
    scores_classifiers['GMM'], LVAL)
actdcfs_fusion, mindcfs_fusion = pieffvsDCFs(calibrated_scores,
    calibrated_labels, logOddsRange)

SVAL_LR, SVAL_SVM, SVAL_GMM = apply_best_classifier(DTR, LTR, SVAL)
calibrated_eval_scores, calibrated_eval_labels = fusion_scores(
    SVAL_LR, SVAL_SVM, SVAL_GMM, LSVAL)
actdcfs_fusion_eval, mindcfs_fusion_eval = pieffvsDCFs(
    calibrated_eval_scores, calibrated_eval_labels, logOddsRange)

plotBayesErrorFusion(logOddsRange, actdcfs_fusion, mindcfs_fusion,
    scores_cal['GMM']["actdcfs"], scores_cal['GMM']["mindcfs"],
    scores_cal['LogReg']["actdcfs"], scores_cal['LogReg']["mindcfs"
    ], scores_cal['SVM']["actdcfs"], scores_cal['SVM']["mindcfs"],
    "Fusion vs Calibrated")
print(f"Best DCF for Fusion: {min(actdcfs_fusion)}, Best minDCF for
     Fusion: {min(mindcfs_fusion)}")
plotBayesErrorFusion(logOddsRange, actdcfs_fusion_eval,
    mindcfs_fusion_eval, eval_scores_cal['GMM']["actdcfs"],
    eval_scores_cal['GMM']["mindcfs"], eval_scores_cal['LogReg']["
    actdcfs"], eval_scores_cal['LogReg']["mindcfs"],
    eval_scores_cal['SVM']["actdcfs"], eval_scores_cal['SVM']["
    mindcfs"], "Fusion vs Calibrated (Evaluation)")
print(f"Best DCF for Fusion (Evaluation): {min(actdcfs_fusion_eval)
    }, Best minDCF for Fusion (Evaluation): {min(
    mindcfs_fusion_eval)}")

# Last point of the evaluation
logOddsRange = np.linspace(-4, 4, 50)
for model in ["full", "diagonal", "tied"]:
    act_dcf_values, min_dcf_values, _ = GMM(DTR, LTR, SVAL, LSVAL,
    model, pi=pi_t)
    plotGMMvsComponents(list(itertools.product([1,2,4,8,16,32],
    repeat=2)), act_dcf_values, min_dcf_values, model)
```

10.3 Results

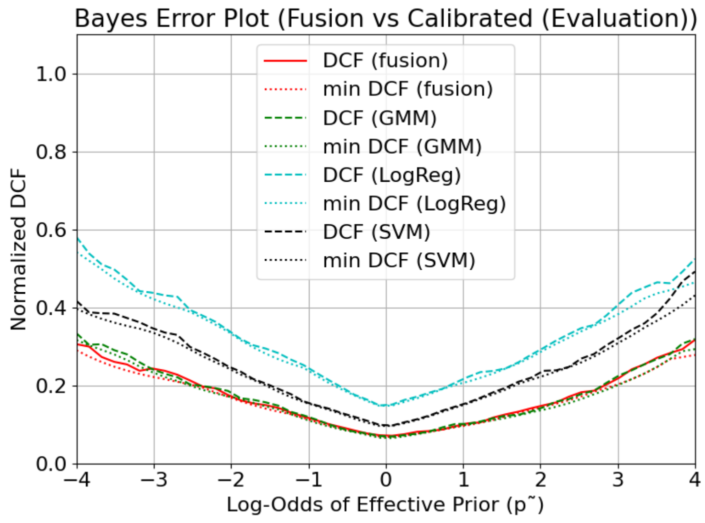


Figure 29: Bayes Error Plot: Fusion vs Calibrated (Evaluation)

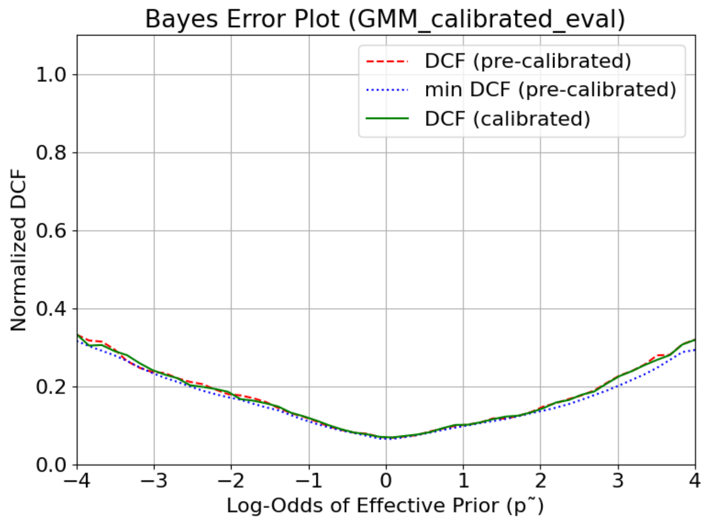


Figure 30: Bayes Error Plot: GMM Calibrated (Evaluation)

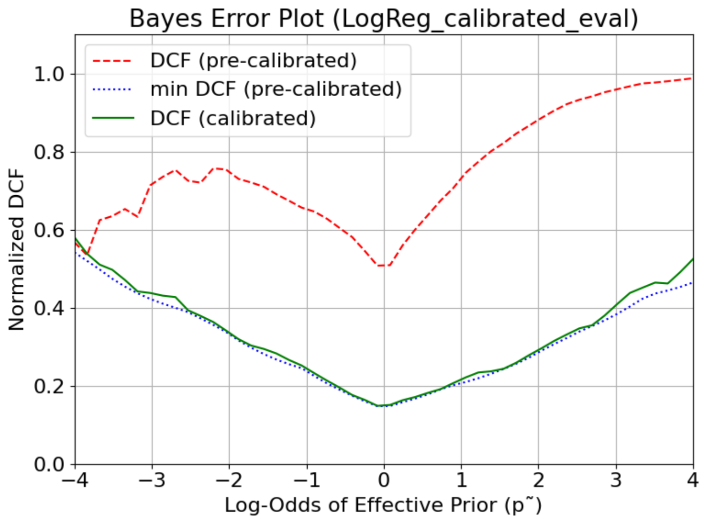


Figure 31: Bayes Error Plot: Logistic Regression Calibrated (Evaluation)

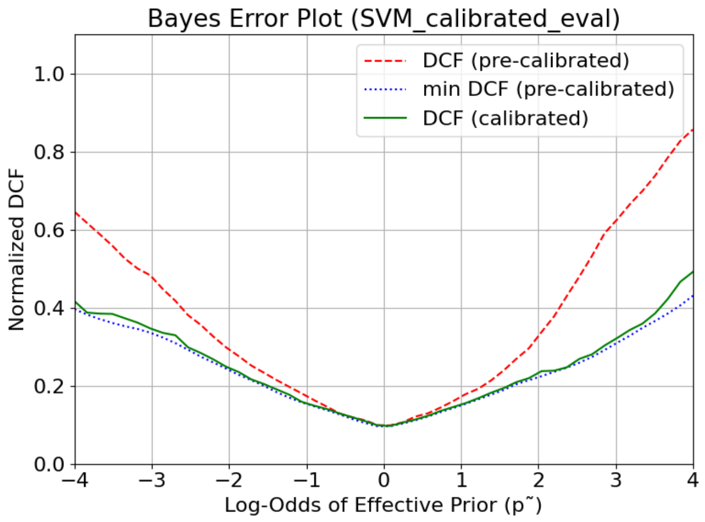


Figure 32: Bayes Error Plot: SVM Calibrated (Evaluation)

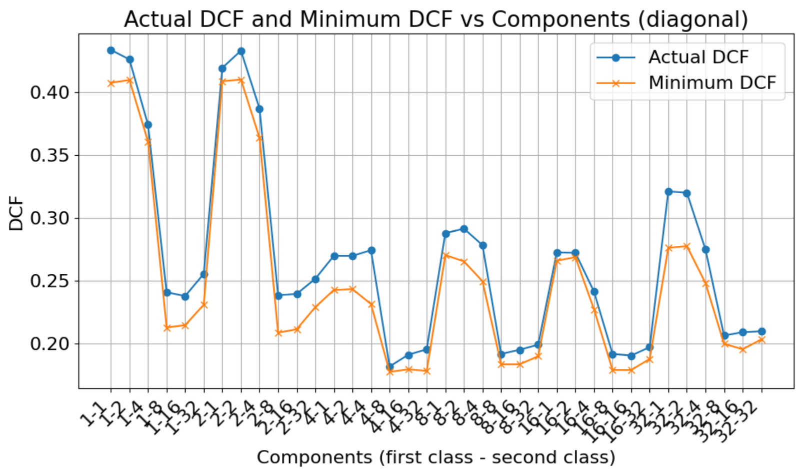


Figure 33: Actual DCF and Minimum DCF vs Components (Diagonal)

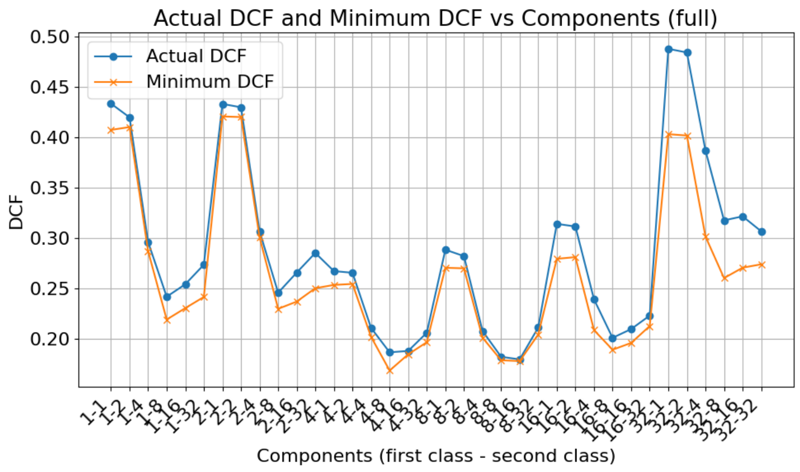


Figure 34: Actual DCF and Minimum DCF vs Components (Full)

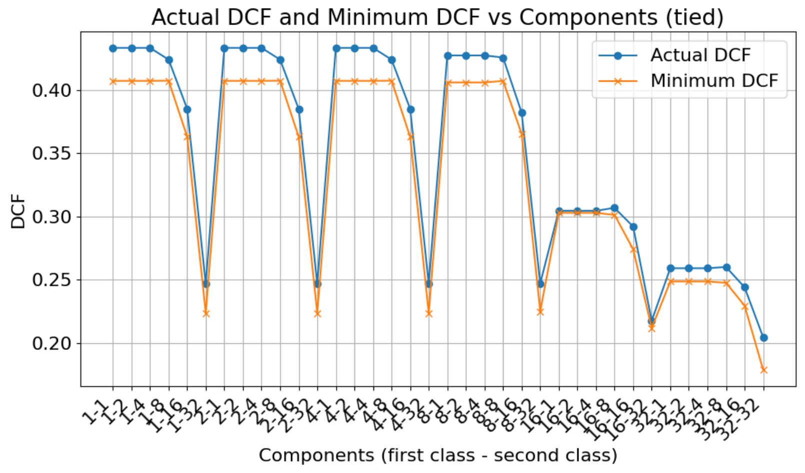


Figure 35: Actual DCF and Minimum DCF vs Components (Tied)

10.4 Analysis of Results

10.4.1 Minimum DCF Comparison

Observing the minDCF of the GMM calculated on the three different models (full, diagonal, tied) and all possible component combinations, and comparing it with the minDCF obtained from the calibrated fusion, we can make the following observations:

- Diagonal Model: The minDCF varies significantly across different component combinations. Some combinations show significantly better performance than others. Combinations with an intermediate number of components tend to provide the best minDCF results.
- Full Model: Similar to the diagonal model, the minDCF varies across combinations, but overall the performance seems less stable. Combinations with a medium to high number of components tend to show lower minDCF.
- Tied Model: The tied model shows a more stable minDCF compared to the diagonal and full models. Combinations with fewer components tend to provide better results, suggesting that higher complexity does not necessarily lead to better discrimination.

10.4.2 Calibrated Fusion Performance

- Fusion Performance: The calibrated fusion shows a minDCF that is comparable to or, in some cases, better than the best combinations of individual GMM models. The fusion benefits from combining the characteristics of each model, mitigating the weaknesses of individual approaches.

- Fusion Advantages: The calibrated fusion offers greater robustness compared to using a single model. While some individual model combinations may have excellent performance, the fusion provides more reliable overall performance. The fusion approach tends to generalize better, as evidenced by the evaluation set performance, indicating better handling of data variations compared to individual models.

10.5 Final Observations

- The calibrated fusion has proven to be an effective approach, combining the strengths of different GMM models and providing a competitive minDCF.
- While some individual GMM component combinations offer excellent performance, the fusion provides a better compromise between complexity and accuracy.
- In a real-world application context, using a calibrated fusion may be preferable to ensure reliable performance across various conditions.

These observations indicate that the calibrated fusion strategy was an effective choice to optimize the classification system's performance, balancing complexity and accuracy effectively.

11 Python Project Code on GitHub

`https://github.com/ale-romeo/ML-Project/`